VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

BOINC server/klient

BOINC Server/Client

2011 Bc. Petr Hanták

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 6. května 2011 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2011 .....

Abstrakt

Práce se zabývá konfigurací a instalací BOINC platformy pro potřeby univerzity. Uvádí do problematiky distribuovaných výpočtů, popisuje základní části platformy a zaměřuje se především na konfigurační části. V praktických příkladech probírá instalaci serveru, vývoj výpočetních aplikací a instalaci klientských aplikací.

Klíčová slova: BOINC, server, klient, distribuované počítání

Abstract

This thesis deals with the configuration and installation of the BOINC platform for the university. Introduces the issue of distributed computing, describes the platform and particular configuration of the parts. In the practical examples, discusses installation of the server, the development of computing applications and how to install client applications.

Keywords: BOINC, server, client, distributed computing

Seznam použitých zkratek a symbolů

BOINC	–	Berkeley Open Infrastructure for Network Computing
CGI	–	Common Gateway Interface protokol
CPU	–	Central Processing Unit
DTD	–	Document Type Definition
HTML	–	Hyper Text Markup Language
HTTP	–	Hypertext Transfer Protocol
PC	–	Personal Computer
PDF	–	Portable Document Format
URL	–	Uniform Resource Locator
SŘBD	–	Systém řízení báze dat
XML	–	eXtensible Markup Language
XSD	–	XML Schema Definition

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V práci se zabýváme problémem distribuovaných výpočtů architektury typu server/klient na bázi softwarové platformy BOINC. Celá problematika je shrnuta do sedmi kapitol. První čtyři kapitoly rozebírají teoretické pozadí a jednotlivé možnosti konfigurací. Zbylé tři kapitoly jsou věnovány praktickým problémům, jako je instalace serveru a výpočetní aplikace.

Kapitola 2 vysvětluje základní pojmy spjaté s platformou BOINC. V podkapitole 2.4 popisujeme, pro které výpočetní aplikace je platforma vhodná.

Technologii XML probírá kapitola 3.1. Bez znalosti základů této technologie není možné provádět jakékoli konfigurace v rámci celé BOINC platformy.

Podrobný popis jednotlivých prvků platformy nalezneme v kapitole 4. Prvkům projektu je věnována celá rozsáhlá podkapitola 4.1. V ní se zabýváme popisem a konfigurací jednotlivých daemonů projektu včetně všech důležitých nastavení konfiguračního souboru projektu. Další důležité části platformy jako jsou Work units a Resulst probírají podkapitoly 4.3 a 4.4.

Jednotlivými kroky instalace serveru nás provádí kapitola 5. Začíná od nutných požadavků na hardware a software, pokračuje kompilací zdrojových souborů BOINC a končí uvedením projektu do provozu. Implementaci a připojení aplikace k projektu popisuje kapitola 6.

Instalaci klientského software můžeme provést podle pokynů v kapitole 7. Užitečnou podkapitolou této kapitoly je bezesporu podkapitola 7.3, která probírá možnosti vzdálené správy dalších klientských aplikací.

Shrnutí výsledků a účelu práce, rozšířené o návrhy možností dalšího vývoje nalezneme v závěrečné kapitole 8.

Zdrojové kódy použité při vývoji, kompilované zdrojové soubory BOINC platformy a VMware image serveru se nachází na přiloženém DVD disku.

2 Co je BOINC?

BOINC (Berkeley Open Infrastructure for Network Computing) je Open-source softwarová platforma vyvinutá na University of California v Berkeley. BOINC vznikl v únoru roku 2002 pod vedením Dr. Davida Andersona. První projekt pro BOINC, Predictor@Home[1], byl spuštěn 9. června 2004.

Účelem BOINC platformy je zajišťovat podporu projektů využívajících Volunteer a grid computing. Nejdříve si vysvětlíme pojem distribuovaný výpočet a následně podrobněji probereme oba výše uvedené pojmy.

2.1 Distribuovaný výpočet

Jedná se o extrémně složitý úkol, který však lze paralelizovat a rozdělit ho tak na více jednodušších částí. Máme-li například program, jehož vstupem je velice rozsáhlá množina vzorků, a tento program musí pro každý ze vzorků provést nějakou simulaci. Pokud bychom tuto vstupní množinu vzorků zpracovávali na jednom procesoru, zabralo by nám to například 1 rok. Kdybychom měli k dispozici procesorů 200 a naši rozsáhlou vstupní množinu mezi ně rozdělili (simulace by probíhaly paralelně), trvalo by nám zpracování celé vstupní množiny vzorků dobu asi 200-krát kratší (v praxi tomu tak není, protože má na dobu výpočtu vliv i např. komunikace mezi procesory, síťová komunikace a další faktory).

2.2 Volunteer computing

Překlad do češtiny nám říká, že jde o dobrovolnické počítání. Prakticky to znamená, že dobrovolníci (většinou členové široké veřejnosti) poskytují své výpočetní zdroje pro projekty založené na distribuovaných výpočtech. Tyto projekty jsou většinou akademické a slouží k výzkumu. Jsou však i výjimky, kdy dobrovolnického počítání využívají projekty nespadající do akademické sféry. První projekt pro dobrovolnické počítání, GIMPS (Great Internet Mersenne Prime Search), byl spuštěn v roce 1996.

Proč jsou tedy dobrovolnické výpočty výhodné? Odpovědí je cena, výkon a dostupnost. Na světě je veliké množství počítačů a jejich počet stále rapidně narůstá. Na to, aby se pro každý velký výzkumný projekt budoval superpočítač, nejsou dostatečné finanční prostředky. Dobrovolnické výpočty tak mohou využít potenciálně obrovský výpočetní výkon za prakticky zanedbatelnou cenu (náklady na servery a jejich provoz). Dobrovolníci poskytují své stanice i patřičné prostředky nutné pro chod těchto stanic zcela dobrovolně. Mezi další klady dobrovolnického je rozšiřování zájmu široké veřejnosti o vědu.

Abychom lépe porozuměli principu volunteer computingu, bylo by vhodné zde definovat některé aspekty vztahu mezi dobrovolníky a projektem:

- Dobrovolníci jsou anonymní. I když jsou žádáni například o emailovou adresu a další informace, rozhodně pak nejsou tyto informace spojovány s jejich identitou v rámci reálného světa.

[1] Domovská stránka projektu Predictor@Home je na adrese: http://predictor.chem.lsa.umich.edu

- Dobrovolníci nejsou odpovědní za projekty. Pokud bude dobrovolník nějakým stylem ovlivňovat výpočty (například změní výsledky), projekt nemůže tyto dobrovolníky nijak stíhat a trestat.
- Dobrovolníci naopak musí věřit projektu, že aplikace které jim projekt poskytuje nijak nepoškodí jejich počítač (jak po stránce hardware, tak po stránce software) a nenaruší jejich soukromí.
- Dále dobrovolníci věří, že projekt pravdivě vypovídá o práci, kterou provádí daná aplikace a i o tom jak budou výsledky využity.
- Dobrovolníci věří, že projekt se řídí správnými bezpečnostními politikami a neumožní své zneužití pro škodlivou činnost jakýchkoliv útočníků.

Shrnuto, provozovatel projektu se stará o anonymitu a bezpečnost dobrovolníků, ti na oplátku poskytují projektu své zdroje.

2.3 Grid computing

Gridů jako takových existuje více druhů. My zde uvedeme nejznámější definici pojmu výpočetní grid, kterou uvedli ve své knize pánové C. Kesselman a I. Foster: „Výpočetní grid je hardwarová a softwarová infrastruktura, která poskytuje spolehlivý, standardizovaný, všudypřítomný a levný přístup ke špičkovým výpočetním službám.“[1]

Grid computing z pohledu BOINC je formou distribuovaného počítání v rámci nějaké organizace (firma, univerzita, atd.), která pro tyto výpočty využívá svých vlastních již existujících výpočetních zdrojů (PC, clustery). Oproti volunteer computingu má toto využití několik výhod. Data se pohybují pouze v rámci dané organizace. Využívá se u projektu jen defaultně malého počtu uživatelských účtů a tvorba dalších účtů je zakázána. Nepředpokládá se manipulace s výsledky (například za účelem získání většího kreditu), a tak není nutné data příliš replikovat. Klientská aplikace může být instalována automaticky na dané počítače a může běžet skrytě. Jedinou nevýhodou oproti volunteer computingu je menší množství dostupných výpočetních zdrojů.

2.4 Aplikace vhodné pro BOINC

BOINC je navržen pro podporu aplikací s velkými výpočetními požadavky, vysokými požadavky na výkon a pro aplikace kombinující obojí. Projekt tak umožňuje přístup k velikému množství teraFLOPů (počtu operací v plovoucí řádové čárce za sekundu) výpočetního výkonu CPU a mnoha terabajtům úložného prostoru. Přístup k volbě aplikace, také záleží na našich potřebách. Trochu jiné požadavky jsou kladeny na projekty a aplikace využívající volunteer computingu a jiné požadavky jsou kladeny na aplikace používající BOINC jako výpočetní grid.

Pro náš účel platí právě druhý případ, využijeme BOINC jako výpočetní grid pro danou organizaci (univerzitu). Tato volba nám umožňuje využít pro implementaci jednotlivých aplikací jakýchkoliv programovacích jazyků. V organizacích můžeme snadno

zajistit podporu daných aplikací na klientských stanicích, na rozdíl od volunteer computingu. Protože používáme k přenosu dat mezi serverem a klientskými stanicemi síť organizace, můžeme si dovolit aplikace přenášející veliké množství dat. Pokud by však přenos probíhal skrz internet, je důležité zvážit, zda se vyplatí (přenosy pomocí Internetu mohou být pomalé a nákladné - příliš mnoho času ztratíme přenosem samotných dat). Naše aplikace musí být dělitelná do paralelních částí, které mezi sebou mají pouze několik nebo nejlépe vůbec žádné závislosti. A na závěr aplikace by měla počítat s tolerancí chyb. Pomocí redundantních výpočtů se dá výskyt chyb značně minimalizovat, ale nelze zaručit naprostou bezchybnost.

Pokud aplikace splňuje požadavky uvedené výše, je možné ji použít pro BOINC.

3 Potřebné technologie

BOINC je založen na jazycích C, C++, PHP a značkovacím jazyce XML. První dva, zejména C++, jsou důležité pro implementaci jednotlivých částí BOINC a vývoj nativních aplikací. Předpokládáme alespoň základní znalost těchto jazyků a nebudeme je zde probírat. Skriptovací jazyk PHP je použit pro webovou podporu projektu. K základnímu zprovoznění výpočetního gridu není znalost jazyka PHP potřebná. Úpravami webové prezentace projektu se zde nebudeme zabývat (má význam pouze při zpřístupnění projektu veřejnosti). Značkovací jazyk XML je naopak pro chod BOINC velice důležitý a nyní si probereme jeho základy, abychom ho byli schopni v BOINC správně využívat.

3.1 XML

XML (eXtensible Markup Language) je značkovací jazyk. Značkovací jazyky jako takové se využívají již po několik desítek let. „Asi prvním známým značkovacím jazykem byl GML (Generalized Markup Language), který vytvořili Charles Goldfarb, Edward Mosher a Raymond Lorie při práci na systému pro uchovávání a následné využití právních textů pro IBM.“[2, strana 13]

Po spojení GML se standardním formátovacím jazykem GenCode vznikl roku 1986 jazyk SGML (Standard Generalized Markup Language). Tento jazyk je velice obecný a má široký záběr v mnoha oblastech. Jazyk pro tvorbu internetových stránek – HTML (Hyper Text Markup Language) je právě aplikací jazyka SGML. Jazyk HTML je omezen přesným počtem značek, chybí možnost definovat si vlastní značky. A proto vznikly jazyky XML a XHTML (mutace pro tvorbu webu). Tyto jazyky byly stvořeny vybráním nejdůležitějších a nejpoužívanějších částí jazyka SGML. Je důležité mít na paměti, že jazyku XML jde o význam částí textu, ne o vzhled, jak je tomu například u jazyka HTML.

Co si můžeme pod tím vším představit? Jazyk XML nám umožňuje členit text podle našich potřeb. Můžeme zpracovávat jakýkoliv text, od jednoduchého katalogu CD, nákupního lístku, až po různé tabulkové hodnoty. Vše bude snadno čitelné a srozumitelné jak pro stroj, tak pro člověka díky hierarchické stromové struktuře XML dokumentu. Velikou výhodou je otevřenost tohoto formátu. Nezávisí na platformě a ani na lokálních úpravách nástrojů. Specifikace XML je všude stejná [5].

I jednoduchou tabulku (1), si můžeme zobrazit velice snadno jako XML dokument.

```
<?xml version="1.0" encoding="utf-8"?>
<skola>
    <student>
        <id>M1021</id>
        <prijmeni>Tišný</prijmeni>
        <jmeno>Jan</jmeno>
        <rocnik>1</rocnik>
    </student>
    <student>
        <id>M1022</id>
        <prijmeni>Malásek</prijmeni>
        <jmeno>Petr</jmeno>
        <rocnik>3</rocnik>
```

```
    </student>
    <student>
        <id>M1023</id>
        <prijmeni>Šťastná</prijmeni>
        <jmeno>Jarmila</jmeno>
        <rocnik>1</rocnik>
    </student>
</skola>
```

Výpis 1: Tabulka studentů v jazyce XML

Z výpisu (1) můžeme vidět, že každý XML dokument začíná nutnou hlavičkou, určující kódování dokumentu a samotnou verzi jazyka XML. Hlavička může obsahovat i další náležitosti, například šablonu značek. Každý dokument dále musí obsahovat právě jeden kořenový element. V našem případě je to element <skola>. Elementy vnořené v tomto elementu mohou mít již libovolný počet duplicit. XML jazyk rozlišuje malá a velká písmena, tedy názvy elementů <Student> a <student> nejsou totožné a budou brány jako dva různé. V názvech elementů se mohou vyskytovat i neobvyklé znaky, například pomlčky a tečky (pokud ale nechceme mít potíže při zpracovaní dokumentu, je lepší se těmto znakům vyhnout).

Značky - tagy - dělíme na párové a nepárové. Párové tagy mohou obsahovat další vnořené elementy na rozdíl od tagů nepárových, které mohou samy o sobě obsahovat pouze množinu atributů. V našem příkladu se vyskytují tagy pouze párové (mají začínající a ukončovací tag). Nepárový tag by vypadal takto: <kos hrusky="10"jablka="10"/>. Tedy element `kos` má dva atributy – `hrusky` a `jablka`, obojí o hodnotě deset. Atributy jsou využívány k upřesnění elementu. Tedy ne každý koš musí obsahovat stejný počet jablek a hrušek. Atributy lze použít i u párových tagů. Tam jsou definovány vždy v počátečním tagu.

Protože XML nám umožňuje využívat neomezené množství elementů s libovolnými názvy, snažíme se většinou vymezit jejich použitou množinu pro dané XML dokumenty pomocí šablon. Tyto šablony (např. DTD a XSD) jasně definují tagy používané dokumentem a umožňují i jeho validaci.

3.1.1 Využití XML

BOINC využívá XML k nastavení veškerých konfigurací na straně serveru (projektu) a na straně klienta. V XML je definován i přenos dat mezi klientem a serverem. Pro tagy v BOINC platí několik základních pravidel.

Os. číslo	Příjmení	Jméno	Ročník
M1021	Tišný	Jan	1
M1022	Malásek	Petr	3
M1023	Šťastná	Jarmila	1

Tabulka 1: Jednoduchá tabulka studentů

Elementy zásadně neobsahují žádné atributy. Pokud elementy nesou nějaké rozhodovací hodnoty (například zákaz registrace uživatelů), pak jsou hodnoty čteny následujícím způsobem. Elementy <tag>1</tag> a <tag /> nesou hodnotu true (pravda). Elementy <tag>0</tag> nesou hodnotu false (nepravda).

Seznam tagů používaných v Boinc není přesně definován pomocí nějakého definičního souboru. Tagy, které BOINC nezná, při parsování daného XML dokumentu jednoduše zahodí. S každou novou verzí software se tagy trochu mění, a tak je vhodné vždy zkontrolovat oficiální dokumentaci BOINC [3].

4 Prvky platformy

V této kapitole si probereme jednotlivé části softwarové platformy BOINC a také se podíváme na jejich vzájemné provázání.

4.1 Projekt

Je základním stavebním prvkem BOINC platformy. Spravuje jednotlivé aplikace pro distribuované výpočty. Projekty jsou vzájemně nezávislé. Každý má svou vlastní databázi, webovou stránku, adresářovou strukturu, aplikace a konfigurační soubory. Na jednom serveru může koexistovat i více projektů najednou.

4.1.1 Master URL

Projekt je identifikován svou vlastní specifickou master URL, která má dvě funkce. Za prvé identifikuje hlavní webovou stránku projektu. Uživatel se tak může dozvědět informace o projektu, navštívit fóra, registrovat si a spravovat svůj účet u tohoto projektu. Druhá funkce master URL slouží klientské aplikaci, BOINC manageru, ke komunikaci s projektem. Přenášejí se například data související se správou a statistikami uživatelského účtu, ale primárně se vyjednává získání nových úkolů (jobs) pro klienta. Jak zjistí klient, že jde o domovskou stránku projektu a že se může dotazovat na jednotlivé pracovní úkoly? Velice jednoduše. V hlavní stránce projektu je obsažen XML element <scheduler >http://*master_url_adresa/cgi/scheduler*</scheduler>. Tento element řekne klientské aplikaci s jakou adresou má přistupovat k deamonu scheduler, o kterém si více řekneme později. Všimněme si, že se ke komunikaci využívá Common Gateway Interface (CGI) protokolu. Ten slouží k propojení webového serveru s externími aplikacemi.

4.1.2 Adresářová struktura projektu

Adresářový strom každého nového projektu můžeme vidět ve výpisu (2). Popišme si, co jednotlivé adresáře obsahují:

- apps - obsahuje spustitelné a další pro běh nutné soubory jednotlivých aplikací a jejich verze
- bin - daemony projektu a další programy (pro generování jobů, verzování aplikací, atd.)
- cgi-bin - CGI programy, které volá klientská aplikace
- log_HOSTNAME - logovací soubory jednotlivých daemonů běžících na straně serveru
- pid_HOSTNAME - uzamčené soubory, pid soubory
- download - místo pro data, která server poskytuje ke stažení

- html - PHP soubory pro veřejnou i privátní část webového rozhranní projektu
- keys - šifrovací klíče generované pomocí OpenSSL
- upload - adresář pro data odesílaná serveru, například výstupní soubory jednotlivých úkolů

```
PROJECT/
    apps/
    bin/
    cgi-bin/
    log_HOSTNAME/
    pid_HOSTNAME/
    download/
    html/
        inc/
        ops/
        project/
        stats/
        user/
        user_profile/
    keys/
    upload/
```

Výpis 2: Adresářová struktura projektu

4.1.2.1 Hiearchické upload a download adresáře

Pro velké projekty je naprosto běžné, že adresáře pro download a upload obsahují statisíce souborů. Bohužel tato skutečnost působí na unixových systémech nemalé problémy. Adresáře se příliš dlouho prohledávají, a to způsobuje značné zpomalení celého serveru.

Boinc tento problém řeší pomocí hierchických adresářů. To znamená, že oba tyto adresáře v sobě obsahují sérii vygenerovaných podadresářů a do nich se soubory pomocí hashovací funkce vždy roztřídí. Hashovací funkce rozděluje soubory na základě jejich jména. Množství adresářů, do kterých se budou soubory rozřazovat, určuje parametr *fanout*. Typicky se tento parametr nastavuje v konfiguračním souboru projektu - *config.xml*. Jeho defaultní hodnota je nastavena po vygenerování nového projektu na hodnotu 1024. Bude se tedy využívat 1024 podadresářů s názvy *0* až *3ff*.

Při vývoji nám toto dělení může působit drobné obtíže. Vývojáři naimplementované programy fungují naprosto bez problému, ale pokud chceme k souborům přistupovat z námi napsaných programů, musíme vždy nějakým způsobem získat jejich jméno. K tomu nám slouží jednoduché funkce viz. výpis (3).

```
// Tato funkce slouží k obsluze download directory.
int dir_hier_path (
    const char* filename, const char* root, int fanout, char* result,
    bool make_directory_if_needed=false
);

// Funkce k obsluze upload directory.
```

```
int  get_output_file_path (RESULT const& result, string& path);

// Nebo druhá možnost pro více souborů.
int  get_output_file_paths (RESULT const& result, vector<string>& );
```

Výpis 3: Obsluha hiearchických adresářů

Funkce pro obsluhu download directory bere jako své vstupní parametry cestu k souboru, absolutní cestu k download adresáři, hodnotu fanout, proměnnou, do které se má výsledná cesta uložit a poslední hodnota udává, zda se má vytvořit daný podadresář, pokud neexistuje. Díky tomu získáme cestu do download adresáře, kam umístíme náš soubor. Tato funkce má hlavní využití při generování nových úkolů, protože vstupní soubory se musí nejdříve přesunout do správného podadresáře v download directory a až poté se může generovat nový úkol.

Funkce pro upload directory jednoduše získá cestu k souboru/souborům přiřazeným k danému výpočetnímu výsledku. Toho využijeme zejména při obsluze našeho vlastního validátoru a assimilátoru, které popíšeme později.

Funkce nejsou jediný prostředek, který můžeme využít k obsluze hiearchických adresářů. Máme k dispozici také dva jednoduché programy, které se spouští přímo v kořenovém adresáři projektu. První se spouští jako dir_hier_path jméno_souboru a vrátí nám absolutní cestu k souboru v hiearchii. Druhý spustíme pomocí dir_hier_move zdorojový_adresář download_adresář fanout. Jednoduše přesune všechny soubory ze zdrojového adresáře do hiearchické struktury download adresáře.

4.1.3 Daemons

Daemon je serverový program, který nepřetržitě běží a obsluhuje určitou část projektu. Projekt se skládá z více daemonů. Některé daemony můžeme implementovat sami, jiní jsou přímo součástí BOINC a spustí se automaticky, když se spouští i ostatní.

4.1.3.1 Scheduler daemon

Scheduler je základním prvkem každého BOINC projektu. Nedá se definovat jako jeden program, v podstatě se jedná množinu CGI skriptů. Plánuje komu a jakou práci přidělit. Obsluhuje komunikaci mezi klienty a serverem. Pokud scheduler neběží, není možné přijímat další práci (tedy pokud již nebyla připravena) a ani není možné odesílat výsledky výpočtů na server. Aby se scheduler nezdržoval přílišnou komunikací s databází projektu, pouze rozdělí, co se má kam poslat a konečné předání nechá na feeder daemonu. Jak jsme se zmínili, scheduler přijímá i výsledky práce od klienta. Informace o výsledcích zanese do databáze, kde jsou připraveny pro zpracování dalšímy daemony. Do implementace scheduleru není třeba nijak zasahovat.

4.1.3.2 Feeder daemon

Feeder je pomocník scheduleru. Izoluje připojení do databáze od CGI skriptů (každý klient má jejich vlastní instanci na serveru). Zpracovává požadavky scheduleru a přiděluje práci klientům. Opět i zde není nutné feeder nějak přizpůsobovat.

4.1.3.3 Transitioner daemon

Tento daemon je velice důležitý. Stará se o stavy jednotlivých work unitů (pracovních jednotek). Jejich funkci si probereme podrobně později. Prozatím nám bude stačit, že work unit zastupuje jeden pracovní úkol posílaný ke klientovi. Transitioner tedy prochází jednotlivé stavy během životního cyklu work unitu. Postupně prochází všechny nedokončené work unity a ptá se na tyto otázky:

- Je work unit připraven k odeslání?
- Už byl daný výsledek obdržen?
- Je výsledek validní?
- Mohu to už smazat?

Zastává tedy větší množství úkolů. Generuje v databázi položky výsledků (results), které je potřeba připravit pro každý work unit. Generuje další výsledky, pokud se stane, že se nám vrátí od klienta chyba při výpočtu. Generuje další výsledky i v případě chyby validace. Dále také transitioner předává výsledky k validaci, pokud máme počet „Success" výsledků u work unitu alespoň tak veliký, jak nám udává kvóta. Dále volá assimilator daemon na zvalidované výsledky. A na závěr volá file deleter daemon.

4.1.3.4 Validator daemon

Porovnává a validuje výstupy daných výsledků u work unitů. Na základě validace přiděluje kredit uživatelům. Musíme spouštět novou instanci pro každou aplikaci, kterou máme u projektu. U validátoru máme několik možností. Můžeme použít validátory z instalace BOINC serveru, nebo si můžeme naprogramovat validátor vlastní. Pokud naše aplikace generuje v případě jednoho workunitu naprosto stejné výsledky (například pokud neprovádí výpočty s desetinnou čárkou, nebo pokud máme povolenu homogenní redundanci), můžeme použít *sample_bitwise_validator*. Ten porovná výsledky bajt po bajtu. Dále je k dispozici *sample_trivial_validator*. Ten prohlásí výsledky za validní, pokud jejich CPU čas překročí určité minimum. To nám umožní sledovat alespoň základní chyby. Oba uvedené programy můžeme spouštět s těmito argumenty (zvolili jsme jen nejdůležitější, zbylé se týkají ovlivňování hodnoty kreditu a je možné je nalézt v dokumentaci):

- – –app appname - povinný atribut, který přiřazuje validátoru danou aplikaci
- (– –one_pass_N_WU N) - zvaliduje maximálně N work unitů a pak se ukončí
- (– –one_pass) - provede jeden průchod nad celou tabulkou work unitů a ukončí se
- (– –mod n i) - zpracuje jen work unity pro které platí (id mod n) == i

Řekli jsme si také o možnosti naprogramovat náš vlastní validátor. K tomu můžeme využít základního validačního frameworku. Musíme implementovat tři funkce z výpisu (4).

```
// první funkce
extern int  init_result (RESULT& result, void*& data);

// druhá funkce
extern int compare_results(RESULT& r1, void* data1, RESULT& r2, void* data2, bool& match);

// třetí funkce
extern int cleanup_result(RESULT& r, void* data);
```

Výpis 4: Funkce pro implementaci validátoru

První funkce vezme výsledek, přečte jeho výstupní soubor a načte si ho do datové struktury. Funkce vrací 0, když je výsledek úspěšný. Když je soubor na vzdáleném médiu, které není přístupné, vrací ERR_OPENDIR a pokusí se jej zpřístupnit později. Jakákoli jiná vrácená hodnota je brána jako chyba a výsledek je dán do stavu "Invalid".

Druhá funkce vezme dva výsledky a porovná jejich načtené datové struktury. Vrací, zda jsou ekvivalentní, či nikoliv.

Poslední funkce uvolní datové struktury pro další použití.

Jednoduchý validátor, který porovnává dva výsledky, můžeme vidět ve výpise (5). Soubory výsledků obsahují jeden integer a jeden double. Validátor zhodnotí výsledky jako ekvivalentní, pokud se hodnota čísel rovná. U double je přípustná tolerance 0,01. Z ukázky je názorně vidět, že validátory se většinou implementují přímo pro danou aplikaci.

```
#include <string>
#include <vector>
#include <math.h>
#include "error_numbers.h"
#include "boinc_db.h"
#include "sched_util.h"
#include " validate_util .h"
using std::string;
using std::vector;

struct DATA {
    int i;
    double x;
};

// funkce pro načtení dat
int  init_result (RESULT const & result, void*& data) {
    FILE* f;
    FILE_INFO fi;
    int i, n, retval;
    double x;

    retval = get_output_file_path (result, fi.path);
    if (retval) return retval;
    retval = try_fopen(fi.path.c_str(), f, "r");
    if (retval) return retval;
    n = fscanf(f, "%d %f", &i, &x);
```

```
    fclose(f);
    if (n != 2) return ERR_XML_PARSE;
    DATA* dp = new DATA;
    dp->i = i;
    dp->x = x;
    data = (void*) dp;
    return 0;
}

// porovnání výsledků
int compare_results(
    RESULT& r1, void* _data1, RESULT const& r2, void* _data2, bool& match
) {
    DATA* data1 = (DATA*)_data1;
    DATA* data2 = (DATA*)_data2;
    match = true;
    if (data1->i != data2->i) match = false;
    if (fabs(data1->x - data2->x) > 0.01) match = false;
    return 0;
}

// uvolnění paměti
int cleanup_result(RESULT const& r, void* data) {
    if (data) delete (DATA*) data;
    return 0;
}

// přidělení kreditu
double compute_granted_credit(WORKUNIT& wu, vector<RESULT>& results) {
    return median_mean_credit(wu, results);
}
```

Výpis 5: Příklad vlastního validátoru

4.1.3.5 Assimilator daemon

Tento daemon sbírá validní výsledky dané aplikace a ukládá je na patřičné místo pro pozdější analýzu. Může je třeba jednoduše zkopírovat na jiné místo v souborovém systému, nebo je může zpracovat a uložit do databáze. Každá aplikace by měla mít svůj vlastní assimilátor.

Pokud nechceme data zapisovat do databáze, můžeme pro naši aplikaci jednoduše použít již vytvořený program *sample_assimilator*. Ten zkopíruje výsledky do adresáře *PROJECT/sample_results/*, kde PROJECT je domovský adresář projektu.

Pokud budeme chtít napsat vlastní asimilátor, je nejlepší vycházet ze zdrojového kódu výše uvedeného programu. Nejdůležitější je implementace funkce:

```
int assimilate_handler(WORKUNIT& wu, vector<RESULT>& results, RESULT& canonical_result);
```

Tato funkce vrací 0, pokud byl work unit označen a zpracován. Vrací DEFER_ASSIMILATION, když má být work unit přeskočen a zavolán později. Toho se dá využít, pokud

chceme od naší aplikace čekat například na všechny výsledky. Jakýkoliv jiný výsledek je brán jako chybný.

4.1.3.6 File deleter daemon

Zajišťuje úklid v souborovém systému. Jedná se o interní program BOINC. Jednoduše prochází databázi a následně maže všechny soubory pro work unity a jejich výsledky, když už nejsou nadále potřeba. Tedy pro work unity, jejichž vstup byl proveden klientskými aplikacemi, výsledky byly zvalidovány a asimilovány.

4.1.3.7 Work generator daemon

Work generátor je speciální daemon, který neustále generuje nové work unity. K čemu je to dobré? Některé aplikace nemají pevnou dávku vstupních dat, ale jejich data jsou průběžně generována například za pomoci nějakého pseudonáhodného generátoru. Work generátor tedy kontroluje počet nezpracovaných výsledků v databázi, dokud jejich počet neklesne pod určitou hodnotu. Pak vygeneruje nové work unity. Samozřejmě záleží jen na nás, jak často ho necháme tuto kontrolu provádět. Work generátor si musíme sami implementovat. Příklad implementace generátoru si ukážeme později i s některými dalšími vylepšeními.

4.1.3.8 Registrace daemonů

Daemony přiřazujeme k projektu v jeho konfiguračním souboru. Přiřazení probíhá jednoduše pomocí konfigurace v jazyce XML.

```
<daemon>
  <cmd> feeder -d 3 </cmd>
  [ <host>doménové_jméno_hostitele</host> ]
  [ <disabled> 0|1 </disabled> ]
  [ <output>název_souboru</output> ]
  [ <pid_file>název_souboru</pid_file> ]
</daemon>
<daemon>
 ...
</daemon>
```

Výpis 6: Konfigurace daemonů

Konfiguraci můžeme vidět ve výpise (6). Elementy v hranatých závorkách jsou nepovinné.

Element <cmd> určuje program, který chceme jako daemon spustit. Parametr d určuje hladinu logování. Pro hodnotu 1 se logují pouze chyby. Při hodnotě 2 se logují i varovná hlášení. A hodnota 3 znamená, že se logují naprosto všechny zprávy včetně informačních.

Nepovinný element <host> určuje, kde má být daemon spuštěn. Defaultně se spouští na hlavním serveru projektu.

Element <output> nese jméno výstupního souboru v *log_HOSTNAME* adresáři projektu. Pokud nebude tento element nastaven, použije se automaticky jako defaultní jméno název aplikace. Pokud však spouštíme více instancí stejného programu, měli bychom logovací výstupy odlišit.

U elementu <pid_file> je situace stejná jako u logovacího výstupu. Akorát se jedná o jméno souboru nesoucího ID processu a ukládá se v adresáři *pid_HOSTNAME*.

4.1.4 Periodické úlohy

Kromě daemonů se v projektu využívá periodických úloh (periodic tasks). Většinou se jedná o krátké programy, které se periodicky na serveru spouští a provádějí například nějakou údržbu. Které úlohy a kdy se mají spouštět definujeme v konfiguračním souboru projektu - *config.xml*. Ve výpise (7) můžeme vidět definici úloh v jazyce XML.

```
<task>
   <cmd> nějaký_task </cmd>
   <output> nějaký_task.out </output>
   <period> 5 min </period>
   [ <host> doménové _jméno_hostitele </host>    ]
   [ <disabled> 0|1 </disabled>  ]
   [ <always_run> 0|1 </always_run> ]
</task>
<!-- Druhý task jako PHP skript -->
<task>
   <cmd> run_in_ops update_forum_activities.php </cmd>
   <output> update_form_activities.out </output>
   <period> 1 day </period>
</task>
<task>
     ...
</task>
```

Výpis 7: Konfigurace periodic tasků

Párový element <cmd> označuje program, který se má spustit. Tento program se musí nacházet v adresáři *PROJECT/bin/*. V druhém případě vidíme, že můžeme spouštět úlohy i jako PHP skript pomocí příkazu run_in_ops název_skriptu. Takový skript musí začínat řádkem: #! /usr/bin/env php.

Element <output> určuje soubor pro výstup skriptu a element <period> udává časovou periodu. Element <host> je nepovinný a říká, kde má být task spuštěn. Defaultně se spouští na hlavním serveru projektu. Dalším nepovinným elementem je <disable>. Tento element nám umožňuje daný task jednoduše zakázat. Prostě nebude prováděn. Posledním nepovinným elementem je <always_run>. Ten říká, že daný task má být spuštěn nehledě na cokoliv (například skript, který loguje zátěž CPU na serveru).

Nový projekt má předdefinovanou řadu úloh. Většina z nich je při použití BOINC jako výpočetního gridu zbytečná. Přesto si je popíšeme, ať víme, které při konfiguraci vypnout:

- db_dump - zapíše statistická data z databáze do XML souboru vhodného k exportu (doporučená perioda je 1 den)
- update_profile_pages.php - generuje HTML soubory se seznamy uživatelských profilů, což je pro výpočetní grid naprosto nepodstatné, protože máme zakázánu registraci uživatelů (doporučená perioda je jednou za pár dní)

- update_stats - updatuje statistiku získaného průměrného kreditu uživatelů, týmů a strojů. Toto nastavení je opět pro grid zbytečností (doporučená perioda je jednou za pár dní).
- update_uotd.php - volba nového uživatele dne je pro nás opět zbytečná (doporučená perioda je 1 den)
- update_forum_activities.php - sleduje aktivity na fóru a dle toho řadí jednotlivá témata. Jelikož nehodláme fórum využívat, je tato volba zbytečná (doporučená perioda 1 hodina)
- team_import.php - importuje seznam týmů z centrální BOINC repository. Pro grid zbytečné.
- notify.php - zasílá přehledové emaily uživatelům (doporučená perioda 1 den)

4.1.5 Konfigurační soubor projektu

Konfigurační soubor projektu se nachází v kořenovém adresáři projektu. Tento soubor se jmenuje **config.xml**. Je generován automaticky při vytvoření nového projektu. Abychom v projektu mohli zprovoznit naše aplikace a aby projekt pracoval dle našich představ, musíme konfigurační soubor přizpůsobit. Editaci se nevyhneme při každém přidání nové aplikace do našeho projektu.

```
<boinc>
  <config>
    <!-- globální konfigurace projektu -->
  </config>
  <daemons>
    <!-- konfigurace jednotlivých daemonů -->
  </daemons>
  <tasks>
    <!-- konfigurace periodických úloh -->
  </tasks>
</boinc>
```

Výpis 8: Základní bloky konfiguračního souboru

Konfigurační soubor je psán v jazyce XML. Jak můžeme vidět ve výpise (8), skládá se ze tří základních bloků. Blok pro konfiguraci projektu, blok pro správu daemonů a blok pro periodické úlohy. Jak se definují bloky pro periodické úlohy a daemony jsme si již ukázali. Nyní se zaměříme na zbylý globální konfigurační blok. Probereme si jednotlivá nastavení a náš vlastní konfigurační soubor si ukážeme až v kapitole zabývající se praktickou konfigurací projektu.

Elementy uvedené v hranatých závorkách bereme jako nepovinné. Také je nutno zdůraznit, že zde nebudou uvedeny všechny možnosti nastavení. Vybrali jsme pouze ty, které nastavení projektu mohou výrazněji ovlivnit. Zbylé možnosti nastavení je možné vyhledat v dokumentaci [3].

4.1.5.1 Základní nastavení

Úplně základní a povinné prvky konfigurace projektu. Bez jejich konfigurace se projekt nepodaří uvést do chodu.

```
<!-- Hlavní -->
<master_url> Master_URL_projektu </master_url>
<long_name> Název_projektu </long_name>
<host> jméno_stanice </host>


<!-- Adresáře projektu -->
<shmem_key> shared_memory_key </shmem_key>
<download_url> http://master_url/download </download_url>
<download_dir> /PROJECT/download </download_dir>
<uldl_dir_fanout> N </uldl_dir_fanout>
<upload_url> http:// master_url/upload</upload_url>
<upload_dir> /path/to/directory </upload_dir>
<cgi_url> http: // adresa/URL </cgi_url>
<log_dir>path</log_dir>
```

- master_url - URL projektu
- long_name - název projektu
- host - jméno stanice na které běží server
- shmem_key - ID sdílené paměti scheduleru (generováno automaticky)
- download_url - URL pro stahování dat ze serveru
- download_dir - absolutní cesta k download adresáři projektu
- upload_url - URL pro upload dat na server
- uldl_dir_fanout - určuje počet podadresářů hiearchické adresářové struktury
- upload_dir - abslolutní cesta k upload adresáři projektu
- cgi_url - URL scheduleru serveru
- log_dir - absolutní cesta k logovacímu adresáři projektu

4.1.5.2 Nastavení pro databázi

Nastavení přístupů k databázi projektu. Základní přístup do databáze je povinnou položkou konfigurace.

```
<!-- Přístup k databázi -->
<db_name> jméno_databáze</db_name>
<db_host> ip_adresa_databáze </db_host>
<db_user> jméno_uživatelského_účtu_k_databázi </db_user>
<db_passwd> heslo_databázového_účtu </db_passwd>
```

```
<!-- Nepovinné elementy -->
[ <replica_db_name> jméno_databáze </replica_db_host> ]
[ <replica_db_user> jméno_uživatelského_účtu_k_databázi </replica_db_user> ]
[ <replica_db_host> ip_adresa_databáze </replica_db_host> ]
[ <replica_db_passwd> heslo_databázového_účtu </replica_db_passwd> ]
```

- db_name - jméno databáze určené pro projekt
- db_host - IP adresa databáze pro projekt (pokud běží lokálně, stačí *„localhost“*)
- db_user - jméno databázového účtu, účet by měl mít jen omezená práva
- db_passwd - heslo databázového účtu
- replica_db - všechny tyto nepovinné elementy nastavují připojení k záložní databázi (obsah hlavní databáze se do ní replikuje)

4.1.5.3 Konfigurace scheduleru

Různá nastavení pro scheduler. Zabývají se převážně plánováním práce a její distribucí.

```
<!-- Nepovinné elementy -->
[ <one_result_per_user_per_wu/> ]
[ <one_result_per_host_per_wu/> ]
[ <max_wus_to_send> N </max_wus_to_send> ]
[ <min_sendwork_interval> N </min_sendwork_interval> ]
[ <max_wus_in_progress> N </max_wus_in_progress> ]
[ <daily_result_quota> N </daily_result_quota> ]
[ <ignore_delay_bound/> ]
[ <dont_generate_upload_certificates/> ]
[ <ignore_upload_certificates/> ]
[ <locality_scheduling/> ]
[ <nowork_skip> 0|1 </nowork_skip> ]
```

- one_result_per_user_per_wu - říká, že v rámci work unitu jsou úkoly pro jednotlivé výsledky zasílány mezi různé uživatele
- one_result_per_host_per_wu - tolerantnější definice, která umožňuje zaslat více výsledků jednomu uživateli, ale jen pokud má více výpočetních zařízení
- max_wus_to_send - říká kolik úkolů bude uživateli maximálně zasláno jako odpověď na dotaz o novou práci (měli bychom to přizpůsobit tak, aby uživatel měl práci i po dobu, kdy nebude například připojen do sítě)
- min_sendwork_interval - časová prodleva mezi odesláním starých výsledků a příjmem nových úkolů (v sekundách)
- max_wus_in_progress - udává, kolik úloh může uživatelský stroj zároveň zpracovávat

- daily_result_quota - kvóta udává, kolik práce na jedno CPU může být zasláno dané klienstské stanici za den (číslo by mělo být opravdu tolerantní)
- ignore_delay_bound - ruší zpoždění odeslání výsledků ukončených úloh (normálně je stanovena menší prodleva)
- dont_generate_upload_certificates - do výsledků se nevkládají nahrávací certifikáty, značně to zrychlí generování výsledků, ale přenos následně není šifrovaný
- ignore_upload_certificates - toto nastavení musí být zapnuto, pokud negenerujeme nahrávací certifikáty (jinak by server data nepřijímal)
- locality_scheduling - pokud je tato možnost zapnuta, scheduler se snaží zadávat klientovi podobnou práci (pokud využívá stejných souborů). Díky tomu se redukuje přenos stejných dat ze serveru.
- nowork_skip - používá se při přetížení databáze. Zajišťuje, že pokud scheduler nemá žádnou práci k rozdávání, neprovádí zbytečné dotazy do databáze.

4.1.5.4 Nastavení pro webovou prezentaci projektu

Toto nastavení není potřeba vůbec požívat, pokud se rozhodneme nevyužívat webovou prezentaci projektu.

```
<!-- Nepovinné elementy -->
[ <profile_screening/> ]
[ <show_results/> ]
[ <no_forum_rating/> ]
[ <users_per_page>N</users_per_page> ]
[ <teams_per_page>N</teams_per_page> ]
[ <hosts_per_page>N</hosts_per_page> ]
[ <profile_min_credit>X</profile_min_credit> ]
[ <team_forums_members_only>0|1</team_forums_members_only> ]
```

- profile_screening - nezobrazují se profilové obrázky, pokud nejsou schváleny adminem
- show_results - povolí zobrazování informací o výsledcích prostřednictvím webu
- no_forum_rating - vypne hodnocení příspěvků na fóru
- users_per_page - počet uživatelů zobrazovaný na stránce
- teams_per_page - počet týmů zobrazovaných na stránce
- hosts_per_page - počet stanic zobrazovaných na stránce
- profile_min_credit - minimální kredit uživatele potřebný k přístupu do editace jeho profilu
- team_forums_members_only - privátní týmová fóra (jen pro členy týmu)

4.1.5.5 Konfigurace pro klienty

Tato nastavení mají význam pouze pokud máme u projektu více uživatelských účtů a nemáme je pod přímou kontrolou. Může tak dojít k situaci, kdy různí uživatelé používají různé verze klientského software. Tyto verze však mohou být starší než verze, které hodláme podporovat. Nastavení uvedená níže nám pomáhají tuto situaci alespoň částečně řešit.

```
<!-- Nepovinné elementy -->
[ < verify_files_on_app_start /> ]
[ <min_core_client_version> N </min_core_client_version> ]
[ <min_core_client_version_announced> N </min_core_client_version_announced> ]
[ <min_core_client_upgrade_deadline> N </min_core_client_upgrade_deadline> ]
```

- verify_files_on_app_start -před spuštěním, nebo po restartu aplikace se kontroluje integrita vstupních dat pomocí MD5 nebo certifikátu. To nám umožňuje rozpoznat manipulaci se soubory.
- min_core_client_version - scheduler se ptá na verzi klienta, ta je mu vrácena jako celočíselná hodnota (výpočet: 10000 * major + 100 * minor + release)
- min_core_client_version_announced - pokud má klient starší verzi software, bude se mu posílat upozornění, že má provést update verze
- min_core_client_upgrade_deadline - čas do kdy má být update software proveden

4.1.5.6 Logování

Globální nastavení hladin logování na serveru. Chceme-li u jednotlivých daemonů mít různé hladiny logování, musíme tento parametr přetížit přímo u konfigurace daemonů.

```
<!-- Nepovinné elementy -->
[ <fuh_debug_level> N </fuh_debug_level> ]
[ <sched_debug_level> N </sched_debug_level> ]
```

- fuh_debug_level - stanoví obecnou hladinu logovacích výpisů(1-chyby, 2-chyby a varování (výchozí), 3-všechny zprávy)
- sched_debug_level - stanoví hladinu logovacích výpisů scheduleru(1-chyby, 2-chyby a varování (výchozí), 3-všechny zprávy)

4.1.5.7 Mazání souborů

Úpravy nastavení pro file deleter.

```
<!-- Nepovinné elementy -->
[ <delete_delay_hours>X</delete_delay_hours> ]
[ <httpd_user>username</httpd_user> ]
```

- delete_delay_hours - file deleter počká daný počet hodin než data smaže

- httpd_user - definuje uživatele pod kterým webový server běží. File deleter pak maže soubory, které náleží jen tomuto uživateli.

4.1.5.8 Další možnosti konfigurace

Zde uvedeme několik možností nastavení, které mohou být užitečné. Zvláště zákaz registrace nových účtů využijeme při používání BOINC pro potřeby organizací.

```
<!-- Nepovinné elementy -->
[ <ended>0|1</ended> ]
[ <disable_account_creation/> ]
[ <min_passwd_length> N </min_passwd_length> ]
[ <request_time_stats_log/> ]
[ <dont_store_success_stderr/> ]
```

- ended - konec projektu, pokud je hodnota 1, posílá se upozornění klientům
- disable_account_creation - vypne možnost založení nového účtu (používá se u gridu)
- min_passwd_length - stanoví požadovaný minimální počet znaků hesel
- request_time_stats_log - scheduler bude od klienta dostávat log, ve kterém budou časové informace o výpočtu
- dont_store_success_stderr - v případě úspěšného výsledku se nebude do databáze zapisovat hodnota standardního chybového výstupu

4.2 Databáze

Databáze běží na SŘBD MySQL. Pro každý projekt se generuje vlastní, výchozí struktura je však stejná. Skládá se z 33 tabulek, které obsahují všechny informace k projektu. Není třeba znát podrobně provázání jednotlivých tabulek. Bude nám stačit popis pouze několika hlavních, s kterými můžeme přijít během vývoje a provozu projektu do styku.

Databázová tabulka **platform** obsahuje informace o jednotlivých výpočetních platformách podporovaných projektem. Po vygenerování projektu je tato tabulka prázdná. Musíme nejdříve specifikovat, které platformy budeme využívat. Můžeme to provést přímo vložením dat do tabulky, nebo můžeme rozšiřovat seznam postupně. Postupného rozšiřování docílíme tak, že při přidávání nové aplikace vždy uvedeme platformy, které podporuje. Pokud tyto platformy nebudeme mít uvedeny v naší tabulce, doplní se (bohužel tento způsob využívá staršího způsobu přidávání aplikací přes *xadd*). Seznam všech platformem, které může BOINC v současné době podporovat, se nachází v tabulce (2).

Základní informace o aplikacích přidaných do projektu nalezneme v tabulce **app**. Nachází se zde celý název aplikace, její zkratka (ta je velice důležitá a odkazujeme se na ni například u konfigurace daemonů) a povolený rozsah verzí.

V tabulce **app_version** nalezneme seznam všech verzí aplikací provozovaných projektem. Každý záznam obsahuje kromě čísla verze a přiřazené aplikace i informace o souborech aplikace. Ty obsahují cestu a MD5 otisk k jednotlivým souborům aplikace. Tyto informace jsou předávány klientům při stahování aplikace.

Informace o uživatelských účtech jsou obsaženy v databázové tabulce **user**. Jednotlivé záznamy obsahují všechny údaje, které uživatel vyplnil při registraci.

Tabulka **host** obsahuje informace o klientských výpočetních stanicích. Tyto informace nemusí sloužit jen jako statistické informace pro uživatele, ale může jich být využíváno i při plánování práce.

Zkratka	**Název**
windows_intelx86	Microsoft Windows (98 or later) running on an Intel x86-compatible CPU
windows_x86_64	Microsoft Windows running on an AMD x86_64 or Intel EM64T CPU
i686-pc-linux-gnu	Linux running on an Intel x86-compatible CPU
x86_64-pc-linux-gnu	Linux running on an AMD x86_64 or Intel EM64T CPU
powerpc-linux-gnu	Linux running on a 32-bit PowerPC processor
ppc64-linux-gnu	Linux running on a 64-bit PowerPC processor
alpha-hp-linux-gnu	Linux running on Alpha
ia64-linux-gnu	Linux running on IA64 (Itanium)
sparc-sun-linux-gnu	Linux running on SPARC
sparc64-sun-linux-gnu	Linux running on SPARC 64-bit
powerpc-apple-darwin	Mac OS X 10.3 or later running on Motorola PowerPC
i686-apple-darwin	Mac OS 10.4+ running on an Intel CPU
x86_64-apple-darwin	Mac OS 10.5+ running on an Intel 64-bit CPU
sparc-sun-solaris2.7	Solaris 2.7 running on a SPARC-compatible CPU
sparc-sun-solaris	Solaris 2.8+ running on a SPARC-compatible CPU
sparc64-sun-solaris	Solaris 2.8+ running on a SPARC 64-bit CPU
hppa-hp-hpux	HPUX running on 32-bit HPPA
hppa64-hp-hpux	HPUX running on 64-bit HPPA
alpha-hp-tru64	Tru64 Unix running on Alpha
ia64-hp-hpux	HPUX running on IA64
powerpc-ibm-aix	AIX running on PowerPC
i686-pc-freebsd	FreeBSD on x86
x86_64-pc-freebsd	FreeBSD on Intel-compatible 64-bit
i686-pc-openbsd	OpenBSD on x86
x86_64-pc-openbsd	OpenBSD on Intel-compatible 64-bit
i686-pc-solaris	Solaris 2.8+ on an Intel x86-compatible CPU
x86_64-pc-solaris	Solaris 2.8+ on an AMD x86_64 or Intel EM64T CPU
i586-pc-haiku	Haiku on an Intel x86-compatible CPU
powerpc64-ps3-linux-gnu	Sony Playstation 3 (Cell processor) running Linux

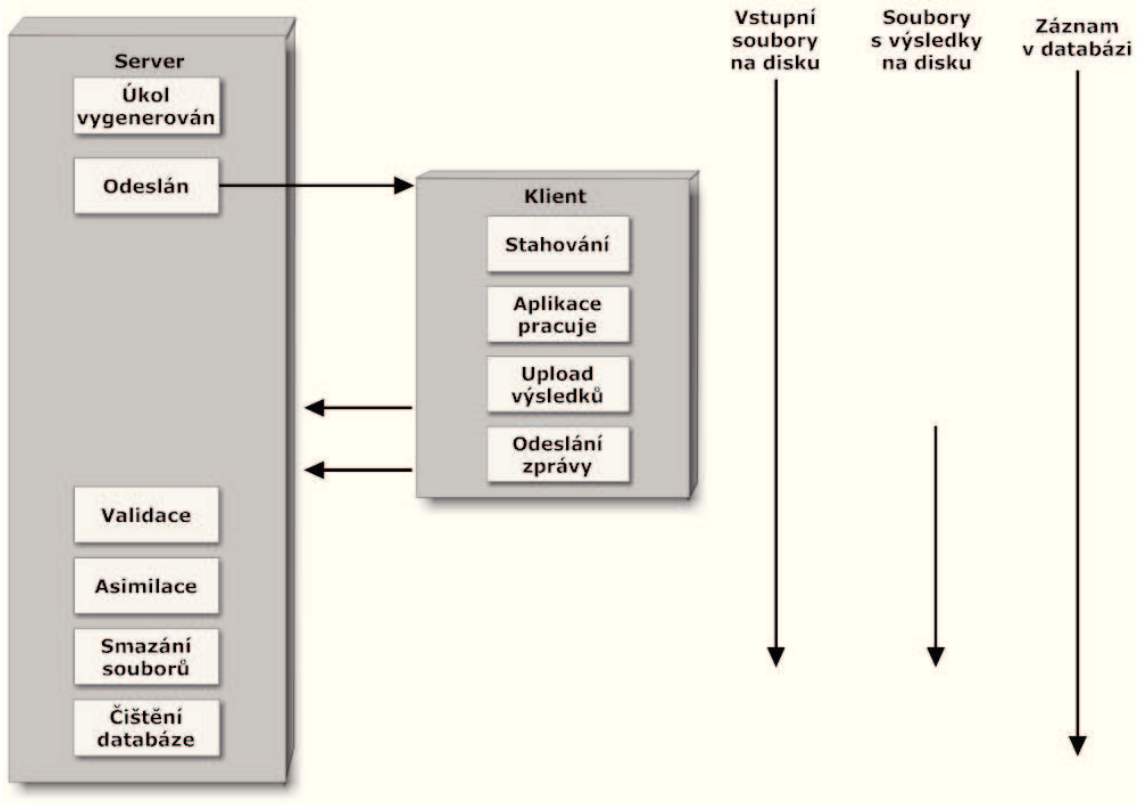
Tabulka 2: Platformy podporované BOINC

Velice důležitou tabulkou je **workunit**. Ta drží informace o jednotlivých úlohách (kde jsou vstupní soubory a na kolik stanic se má úloha rozeslat). Do této tabulky se generuje veliké množství záznamů. Abychom předešli zaplnění, periodická úloha **db_dump** sama odstraňuje záznamy z této tabulky. Odstraňovány jsou work unity, které byly dokončeny před několika týdny.

Poslední důležitou tabulkou je **result**. Zde jsou informace o jednotlivých dílčích úlohách vygenerovaných pro jednotlivé work unity. Čištění této tabulky probíhá stejným způsobem jako u tabulky workunit.

4.3 Work units

Work unit můžeme přeložit jako pracovní jednotku. Když budeme nějakou aplikaci BOINC projektu považovat za normální program, pak work unit obsluhuje jeden průběh této aplikace s jejími vstupy i výstupy. Životní cyklus každé work unit můžeme vidět na obrázku (1). Obrázek platí pouze pro work unit bez redundantních výpočtů.

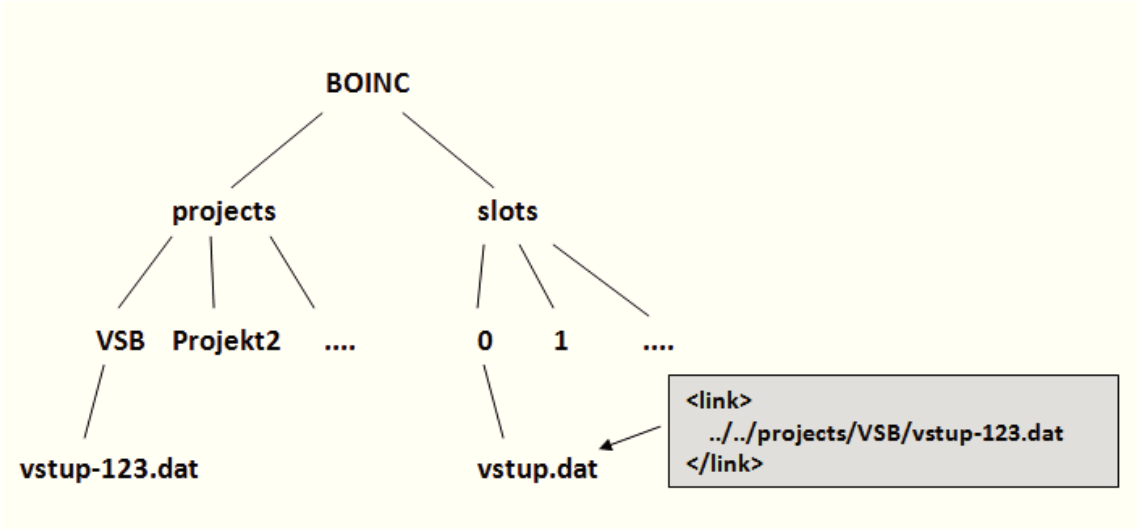


Obrázek 1: Životní cyklus work unitu

Každý work unit se skládá ze vstupní a výstupní šablony. V těch je popsáno, jaké má aplikace vstupy a výstupy. Dále šablony definují co se s těmito soubory má dělat.

Při zadávání práce jsou vstupní soubory přenášeny do adresáře projektu klienta tak, jak jsou uloženy na serveru. Ale aplikace většinou načítá nějaké soubory a může požadovat, aby se jmenovaly stejným jménem. Například vstupní soubor *vstup-123.dat* by se měl jmenovat *vstup.dat*. Abychom toto přejmenování zabezpečili, definuje se ve vstupní šabloně i logický název daného souboru. Jakmile se má aplikace u klienta spustit, klientská aplikace poupraví názvy na logické a předá je jako vstup aplikaci. Ta může následně bezchybně pracovat. Na obrázku (2) vidíme jak tento systém funguje. Složka slots obsahuje jednotlivé běžící aplikace. Jeden slot pro každý procesor.

Šablony jsou ukládány na serveru v adresáři *PROJECT/templates*. *PROJECT* je domovský adresář projektu.



Obrázek 2: Adresářová struktura na klienstké stanici

4.3.1 Vstupní šablona

Šablona se zaměřuje především na vstupní soubory. Definuje, jaké mají mít logické jméno, zda se mají kopírovat nebo zda mají po skončení výpočtu zůstat u klienta. Také je možnost v šabloně definovat požadavky na výkon a paměť. Tyto požadavky se mohou uvést přímo v šabloně, ale pokud používáme ke generování nových úloh work generátor, je mnohem praktičtější definovat tyto požadavky v programu work generátoru. V takovém případě šablona obsahuje pouze reference na vstupní soubory.

```
< !-- Informace k souboru -->
< file_info >
    <number>0</number>
    < !-- Nepovinné elementy -->
    [ <sticky/> ]
    [ <nodelete/> ]
</ file_info >
```

```
<workunit>
    < !−− Definice výše uvedeného souborů −−>
    < file_ref >
        <file_number>0</file_number>
        <open_name>Logické jméno</open_name>
        < !−− Nepovinný element −−>
        [ <copy_file/> ]
    </ file_ref >
    < !−− Nepovinné elementy −−>
    [ <command_line>−flags xyz</command_line> ]
    [ <rsc_fpops_est>x</rsc_fpops_est> ]
    [ <rsc_fpops_bound>x</rsc_fpops_bound> ]
    [ <rsc_memory_bound>x</rsc_memory_bound> ]
    [ <rsc_disk_bound>x</rsc_disk_bound> ]
    [ <delay_bound>x</delay_bound> ]
    [ <min_quorum>x</min_quorum> ]
    [ <target_nresults>x</target_nresults> ]
    [ <max_error_results>x</max_error_results> ]
    [ <max_total_results>x</max_total_results> ]
    [ <max_success_results>x</max_success_results> ]
</workunit>
```

- file_info - definuje blok pro každý vstupní soubor
- number - pořadové číslo vstupního souboru
- sticky - soubor zůstane u klienta i po ukončení výpočtu
- nodelete - soubor nebude po ukončení výpočtu smazán ze serveru
- workunit - blok pro definici konfigurace work unit
- file_ref - blok pro rozšířenou konfiguraci vstupního souboru
- open_name - logické jméno vstupního souboru
- copy_file - soubor bude u klienta zkopírován přímo do daného slot adresáře
- command_line - argumenty předané hlavnímu programu aplikace
- rsc_fpops_est - odhad průměrného počtu FLOPů pro provedení výpočtu
- rsc_fpops_bound - maximální počet FLOPů na celý výpočet, při překročení výpočet selže
- rsc_memory_bound - odhadovaná velikost operační paměti vymezená pro výpočet, práce bude zasílána jen klientům, kteří mají dostatečnou velikost operační paměti
- rsc_disk_bound - odhadovaná velikost diskového prostoru zabraná aplikací i konkrétním výpočtem dohromady, práce bude zasílána jen klientům s dostatečnou kapacitou

- delay_bound - čas vymezený pro práci (v sekundách)
- min_quorum - minimální počet Success výsledků
- target_nresults - počet rozesílaných výsledků
- max_error_results - maximální počet chybných výsledků, při překročení hodnoty je work unit prohlášen za chybný
- max_total_results - maximální počet přijatých výsledků při překročení hodnoty je work unit prohlášen za chybný
- max_success_results - maximální počet úspěšných výsledků, při překročení hodnoty je work unit prohlášen za chybný

Je vhodné upravit počet jednotlivých vstupních souborů v rámci jednoho úkolu tak, aby jich nebylo příliš mnoho. Soubor vstupní šablony se doplní o patřičné údaje a následně se vloží do databáze jako záznam typu BLOB. Pokud by tedy byl soubor příliš veliký, nemusel by se nám do databáze vlézt (samozřejmě záleží taky na nastavených kvótách). V dokumentaci je doporučeno, aby byl počet vstupních souborů menší než 100. Úplně nejvhodnější je archivovat více vstupních souborů do jednoho či více ZIP souboru/ů.

4.3.2 Výstupní šablona

Výstupní šablona má podobné základní prvky jako šablona vstupní, ale zaměřuje se na soubory odesílané zpět na server.

```
< file_info >
    <name><OUTFILE_0/></name>
    <generated_locally/>
    <upload_when_present/>
    <max_nbytes>číslo</max_nbytes>
    <url><UPLOAD_URL/></url>
</ file_info >
<result>
    < file_ref >
        <file_name><OUTFILE_0/></file_name>
        <open_name>jméno_výstupního_souboru</open_name>
        < !-- Nepovinné elementy -->
        [ <copy_file>0|1</copy_file> ]
        [ <optional>0|1</optional> ]
    </ file_ref >
</result>
```

- generated_locally - říká, že je soubor generován lokálně u klienta
- upload_when_present - soubor se po ukončení výpočtu odešte na server
- max_nbytes - maximální velikost výstupního souboru v bajtech, při překročení se neodešle

- url - bude automaticky doplněno
- result - blok pro specifikaci výsledku
- copy_file - soubor bude po ukončení výpočtu ze *slot* adresáře zkopírován do adresáře projektu u klienta
- optional - pokud tento element chybí nebo má hodnotu nula, musí být výstupní soubor vytvořen, jinak výpočet selže

4.3.3 Výstupní šablona

Pracovní jednotky jsou přidávány do databáze generováním. Generování nových jednotek může zajišťovat nějaký work generátor (pokud nemáme připravenu nějakou dávku), nebo můžeme volat program **create_work**. Program se nachází v adresáři *PROJECT/-bin/create_work*, kde *PROJECT* je kořenový adresář projektu. Volání programu *create_work* vypadá následovně:

```
create_work [ argumenty ] vstupní_soubor_1 ... vstupní_soubor_n
```

Program má několik argumentů, které si popíšeme. Argumenty v závorkách jsou nepovinné:

- – –app zkratka_aplikace - povinný atribut, který přiřazuje work unit k dané aplikaci
- (– –wu_name název) - název work unitu (defaultně se generuje *aplikace_PID_čas*)
- (– –wu_template název_souboru) - cesta a jméno vstupní šablony, definováno relativně ke kořenovému adresáři projektu (defaultně *templates/aplikace_in*)
- (– –result_template název_souboru) - cesta a jméno výstupní šablony, definováno relativně ke kořenovému adresáři projektu (defaultně *templates/aplikace_out*)
- (– –priority n) - nastaví u work unitu prioritu, feeder pak upřednostňuje work unity s vyšší prioritou

4.4 Results

Results (výsledky) se vytváří v databázi po přidání nového work unitu. Work unit může mít několik redundantních výsledků.

Každý výsledek prochází na straně serveru několika stavy:

- Init - jedná se o počáteční stav, výsledek byl pouze vytvořen
- Unsent - práce byla zaslána na klientskou stanici, ale zatím stále nebyly dodány data nazpět serveru
- Unsent (in work seq) - podobný stav jako stav předchozí s tím rozdílem, že výpočet právě probíhá

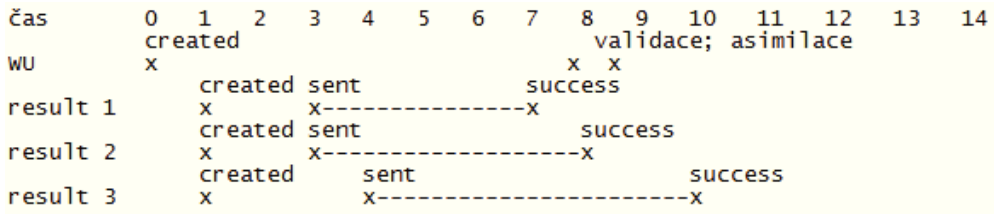
- In Progress - výsledek je zpracováván serverem
- Over - konečný stav

4.4.1 Redundance

Redundance v BOINC znamená provádění jednoho úkolu vícekrát. Protože data putují sítí (Internetem) a výpočetní zdroje jsou poskytovány dobrovolně (neplatí u výpočetního gridu), nemůžeme zaručit správnost každého výpočtu. Z toho důvodu provádíme redundantní výpočty.

Kolikrát se bude stejný výpočet provádět uvádíme ve vstupní šabloně work unitu. Na obrázku (3) vidíme příklad pro nastavení šablony:

- min_quorum = 2
- target_nresults = 3
- delay_bound = 10



Obrázek 3: Redundance results u work unitu

Vidíme, že work unit byl vytvořen v čase 0. V čase jedna transitioner daemon vygeneroval dle nastavení 3 výsledky. V čase 3 byly odeslány první dva výsledky (úkoly) rozdílným klientským stanicím, zbývající byl odeslán v čase 4. Stanice prováděly výpočty. V čase 7 je úspěšně odeslán první výsledek zpět na server. Jakmile přišel i druhý úspěšný výsledek v čase 8, byla splněna minimální kvóta a mohla být provedena validace a asimilace. Třetí výsledek, který dorazil až v čase 10, už neměl na nic vliv. Kdyby se však něco stalo s výsledkem číslo 2, například by nebyl spočítán do času 10, pak by byl použit třetí výsledek. Work unit by byl stále zpracován v pořádku a s minimální časovou prodlevou.

Redundace je pojistkou proti chybám v průběhu výpočtu. Je však na zvážení, kolik výsledků pro jeden work unit hodláme rozesílat. Každý výsledek navíc je na úkor volné výpočetní kapacity pro ostatní work unity.

Doposud jsme si popisovali klasickou redundanci. Může se však stát, že je naše aplikace velice choulostivá na přesnost výpočtu. Pro takové případy máme tzv. **Homogenní redundanci (Homogenous redundancy)**. Ta zajistí, že jednotlivé výsledky work unitu budou zasílány pouze na klientské stanice se stejnou konfigurací.

4.5 Aplikace pro výpočty

Každá výpočetní aplikace pro BOINC musí být zkompilována pro platformu, na které hodláme výpočty provádět. Seznam dostupných platforem je v tabulce (2).

Krom samotného výpočtu má výpočetní aplikace za úkol průběžně informovat klientskou aplikaci o svém postupu (tzv. **fraction state**). Klientská aplikace na základě této informace zobrazuje průběh v progress baru a odhaduje čas do ukončení výpočtu. Fraction state má pouze informativní význam. Pokud se rozhodneme, že není potřeba, můžeme jeho implementaci zanedbat. Na výpočet samotný to nebude mít vliv.

Existuje ještě jedna úloha, kterou musí výpočetní aplikace vykonávat. Průběžně během výpočtu musí do nějakého souboru zaznamenávat, jakou část výpočtu už má za sebou (tzv. **checkpoint state**). Pokud bude činnost BOINC na klientské stanici přerušena (například restartováním stanice), může díky tomuto souboru naše výpočetní aplikace navázat na výpočet a ten nemusí probíhat celý znovu od začátku.

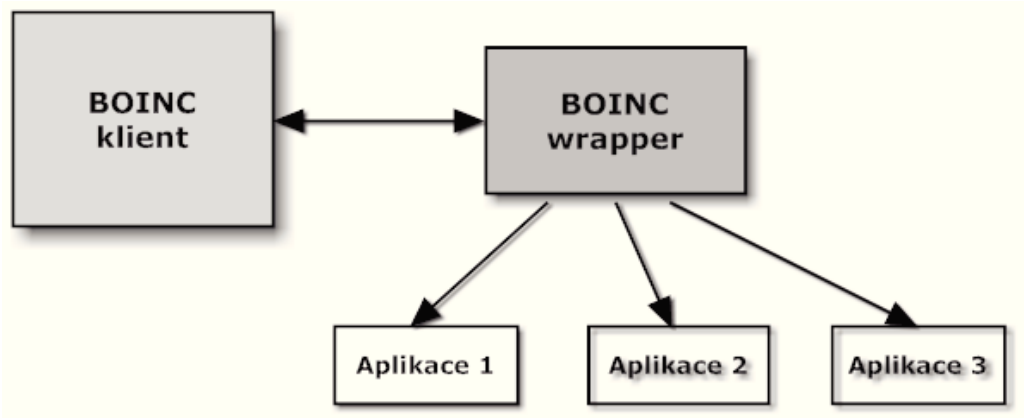
Aplikace mohou být vyvíjeny dvěmi způsoby - jako nativní aplikace nebo jako aplikace využívající wrapperu.

Vývoj nativních aplikací je výhodný, pokud chceme využít některých pokročilejších funkcí (vývoj aplikací s grafickým výstupem pro BOINC screen saver). Jeho nevýhodou je, že musíme sami ošetřit různé stavy mezi klientskou aplikací a naší výpočetní aplikací.

Z toho důvodu je pro vývoj standardních aplikací doporučován wrapper, který ošetřuje tyto stavy za nás. Nyní si probereme jak wrapper funguje a jak jej nakonfigurovat.

4.5.1 BOINC wrapper

Může spustit jakoukoli existující konzolovou aplikaci. Wrapper za nás sám vyřizuje komunikaci s klientskou aplikací (obrázek 4). My pouze jednoduše nadefinujeme, které aplikace má wrapper spustit.



Obrázek 4: Vztah mezi wrapperem a klienstkou aplikací

Zdrojové kódy pro wrapper nalezneme v instalačním balíčku BOINC v adresáři *samples/wrapper/*. Hodláme-li provádět výpočty na operačním systému Windows, můžeme ke kompilaci využít připravený solution pro Visual Studio. Ten se nachází v instalačním balíčku BOINC v adresáři *win_build*.

Většinou nám ve zdrojového kódu wrapperu stačí jednoduchá úprava na řádku 54. Změníme pouze název definičního XML souboru wrapperu na název, který budeme používat u naší aplikace. Následně stačí provést kompilaci.

V definičním souboru wrapperu uvádíme, které aplikace a v jakém pořadí se mají spustit. Všechny tyto soubory uložíme na serveru do adresáře */PROJECT/apps/zkratka_aplikace/zkratka_verze_platforma/*. Do koncového adresáře přesuneme soubory spouštěných aplikací, wrapper a jeho definiční soubor. Název spustitelného souboru wrapperu musí být shodný se jménem adresáře, ve kterém se nachází, tedy *zkratka_verze_platforma*.

Nyní se podíváme na obsah definičního souboru wrapperu:

```
<job_desc>
    <!-- Definice jedné aplikace spouštěné wrapperem -->
    <task>
        <application>moje_aplikace.exe</application>
        <!-- Nepovinné elementy -->
        [ <stdin_filename>stdin_file</stdin_filename> ]
        [ <stdout_filename>stdout_file</stdout_filename> ]
        [ <stderr_filename>stderr_file</stderr_filename> ]
        [ <command_line>--foo bar</command_line> ]
        [ <weight>X</weight> ]
        [ <checkpoint_filename>filename</checkpoint_filename> ]
        [ <fraction_done_filename>filename</fraction_done_filename> ]
        [ <exec_dir>dirname</exec_dir> ]
        [ <setenv>VARNAME=VAR_VALUE</setenv> ]
        [ <daemon/> ]
        [ <append_cmdline_args/> ]
    </task>
    <!-- Zde mohou následovat další aplikace spouštěné wrapperem -->
</job_desc>
```

- task - blok pro definici jedné dílčí aplikace, aplikace se spouští v pořadí dle těchto bloků
- application - název souboru spouštěné aplikace
- stdin_filename - logické jméno souboru posílaného jako standardní vstup
- stdout_filename - logické jméno souboru braného jako standardní výstup
- stderr_filename - logické jméno souboru braného jako standardní chybový výstup
- command_line - argumenty příkazové řádky předané této aplikaci
- weight - udává poměr náročnosti vůči dalším aplikacím

- checkpoint_filename - název souboru pro checkpoint state (pokud ho aplikace používá)
- fraction_done_filename - název souboru pro fraction state (pokud ho aplikace používá)
- exec_dir - adresář, ve kterém se má aplikace spustit (relativní cesta k danému *slot* adresáři)
- setenv - nastaví proměnné prostředí
- daemon - definuje, zda aplikace běží jako daemon. Bude běžet na pozadí po celou dobu, co se budou vykonávat ostatní aplikace běžící normálně.
- append_cmdline_args - připojí argumenty příkazové řádky předané wrapperu v definici work unitu

5 Instalace serveru

Pro server je třeba zvolit vhodnou distribuci operačního systému Linux (Debian, Ubuntu, Fedora a jiné). V našem případě jsme zvolili virtuální obraz systému Fedora 13, běžící na virtualizačním software VMware[2]. Výhodou je snadná přenositelnost.

Při instalaci jsme vycházeli z postupu popsaného v článku pana Myerse [4] a z oficiální BOINC dokumentace [3].

5.1 Požadavky na hardware

- statická IP adresa
- procesor dvoujádrový Xeon nebo Opteron
- 2 GB operační paměti
- 40 GB volného prostoru na pevném disku

Je třeba podotknout, že tato konfigurace je doporučena pro běžné veřejné projekty, které mají tisíce klientských stanic a běží na nich současně desítky aplikací. Berme ji proto s rezervou.

5.2 Uživatelské účty

Pro server je vhodné vytvořit i další uživatelský účet a novou skupinu. Není vhodné, aby byly všechny operace prováděny pod účtem *root*.

My jsme si vytvořili účet *basic* a skupinu *boinc* (obsahuje uživatele: *basic, apache, mysql, root*). Pod tímto účtem budeme provádět veškeré operace nevyžadující uživatelská práva uživatele *root*.

Příkazy prováděné uživatelem *root* budou v tomto textu označeny znakem #.

5.3 Požadavky na software

Aby bylo možné BOINC server zprovoznit, musíme nejdříve zajistit potřebnou softwarovou podporu, kterou je instalace potřebných balíčků Fedory, instalace HTTP serveru Apache a instalace databázového serveru MySQL.

5.3.1 Požadované balíčky

BOINC server ke svému instalaci a provozu požaduje instalaci balíčků s minimální verzí uvedenou zde:

- subversion
- make 3.79+

[2] Více na stránce: http://www.vmware.com/cz/

- m4 1.4+
- libtool 1.5+
- autoconf 2.58+
- automake 1.8+
- GCC 3.0.4+
- pkg-config 0.15+
- Python 2.2+ s MySQLdb module 0.9.2+
- MySQL 4.0.9 a vyšší
- Apache s mod_ssl a podporou PHP5+
- PHP5
- php_gd
- OpenSSL 0.9.8+

5.4 Inicializace služeb potřebných k instalaci

Pokud se nám podařilo balíčky nainstalovat, potřebujeme nyní spustit MySQL server. Ten je potřeba už při vytváření projektu. Server uvedeme do chodu a nastavíme automatické spouštění pomocí příkazů:

```
# chkconfig mysqld on
# service mysqld restart
```

5.5 Zdrojové kódy BOINC

Nejdříve stáhneme zdrojové kódy a provedeme jejich kompilaci. Toho dosáhneme provedením příkazů:

```
svn co http://boinc.berkeley.edu/svn/branches/server_stable
cd server_stable
./_autosetup
./configure --disable-client
make
```

1. Nejdříve se ze subversion repository stáhly zdrojové kódy a uložily se do adresáře *server_stable* v našem aktuálním adresáři.
2. Vstoupili jsme do adresáře *server_stable*.
3. Spustili jsme skript *_autosetup*.

4. Pomocí skriptu *configure* jsme připravili kódy ke kompilaci pouze pro software serveru.

5. Pomocí programu *make* jsme zdrojové soubory zkompilovali.

Tyto zkompilované soubory je vhodné si uložit, protože nám mohou pomoci při vývoji našich vlastních daemonů.

5.6 Nový projekt

Stále se nacházíme ve stejném adresáři se zdrojovými kódy. K vytvoření nového BOINC projektu použijeme jednoduchý program z adresáře *tools*:

```
# cd tools
# ./make_project --project_root /project/boinc boinc VSB@Home
```

Volání programu má tuto strukturu:

```
make_project [nastavení] jméno_projektu [ 'Dlouhé_jméno_projektu' ]
```

Možná nastavení:

- --srcdir - cesta k adresáři se zdrojovými kódy (defaultně . nebo ..)
- --project_root - cesta ke kořenovému adresáři projektu (defaultně *~/projects/jméno_projektu*)
- --key_dir - cesta k adresáři klíčů (defaultně *PROJECT_ROOT/keys*)
- --no_query - přijímá všechny zadané adresáře bez potvrzovacích dotazů
- --delete_prev_inst - smaže předchozí kořenový adresář, pokud existuje
- --url_base - základní URL (defaultně *http://HOSTNAME/*)
- --html_user_url - master URL projektu (defaultně *URL_BASE/jméno_projektu/*)
- --html_ops_url - URL administrátorského rozhranní (defaultně *URL_BASE/PROJECT_NAME_ops/*)
- --cgi_url - CGI URL (defaultně *URL_BASE/jméno_projektu_cgi/*)
- --db_host - adresa databázového serveru (defaultně *localhost*)
- --db_name - jméno databáze (defaultně *jméno_projektu*)
- --db_user - jméno databázového uživatele (defaultně *root*)
- --db_passwd - heslo pro přístup do databáze (defaultně žádné)
- --drop_db_first - pokud existuje databáze se stejným jménem, potom se smaže

Nyní máme v adresáři */project/boinc* vygenerován nový projekt.

5.7 Webový server

Máme vytvořený projekt, ale abychom mohli k projektu přistupovat, musíme mít nakonfigurován a spuštěn webový server. Webový server máme už nainstalován, musíme mu pouze přidat reference na náš projekt. Reference přidáváme do konfiguračního souboru webového serveru: */etc/httpd/conf/httpd.conf*

Co se má vložit nalezneme v automaticky vygenerovaném souboru: *PROJECT/boinc.httpd.conf*

V našem případě vložíme:

```
## Settings for BOINC project VSB@Home

    Alias /boinc /project/boinc/html/user
    Alias /boinc_ops /project/boinc/html/ops
    ScriptAlias /boinc_cgi /project/boinc/cgi-bin

    # Note: projects/*/keys/ should NOT be readable!

    <Directory "/project/boinc/html">
        Options Indexes FollowSymlinks MultiViews
        AllowOverride AuthConfig
        Order allow,deny
        Allow from all
    </Directory>

    <Directory "/project/boinc/cgi-bin">
        Options ExecCGI
        AllowOverride AuthConfig
        Order allow,deny
        Allow from all
    </Directory>
```

Na adrese *https://adresa_serveru/boinc_ops* nalezneme jednoduché administrační rozhranní projektu. Umožňuje prohlížet seznamy uživatelů, work unitů, výsledků,uživatelských stanic a aplikací. Abychom k tomuto nastavení mohli přistoupit, musíme nakonfigurovat přístup chráněný heslem.

Toho docílíme přidáním souboru *.htpasswd* do složky *PROJECT/html/ops* pomocí příkazů:

```
cd html
cd ops
htpasswd -c .htpasswd uživatelské_jméno
```

Zbývá nastavit automatické spouštění serveru a provést restart pro načtení nové konfigurace příkazy:

```
#   chkconfig httpd on
#   service httpd restart
```

5.7.1 Přístupová práva

Při generování projektu byly u některých adresářů nastavena nevyhovující uživatelská práva. Provedeme tedy změnu práv pro následující adresáře a všechny jejich podadresáře:

```
# chown -R basic:boinc /project/boinc

chmod -R 02770 upload
chmod -R 02770 html/cache
chmod -R 02770 html/inc
chmod -R 02770 html/languages
chmod -R 02770 html/languages/compiled
chmod -R 02770 html/user_profile
```

Převedli jsme adresář projektu na uživatele *basic*, následně jsme nastavili potřebná přístupová práva. Protože je uživatel *apache* součástí skupiny *boinc*, neměli bychom mít žádné problémy. Kdyby přece jen nějaké problémy nastaly, můžeme změnit skupinu výše uvedených adresářů na *apache*.

5.8 Databázový server

Doposud jsme měli databázový server v základním nastavení po jeho instalaci, což bylo výhodné z hlediska snadnější tvorby projektu. Nyní musíme zabezpečit přístupy do databáze různými účty a hesly.

Nastavíme heslo k hlavnímu databázovému účtu:

```
# mysqladmin -u root password (nějaké_heslo)
```

Tento účet nebudeme běžně využívat, výhodnější je si založit nějaké další účty. V našem případě si založíme účty *boincadm* a *basic*. Účet *boincadm* bude mít všechna práva nad databází projektu, ale bude využíván pouze jako servisní účet. Účet *basic* bude využíván pro běžné dotazování nad databází projektu.

```
# mysql -p -u root
Password: heslo_hlavního_účtu
mysql> use boinc;
mysql> GRANT SELECT, INSERT, UPDATE, DELETE ON * TO 'basic' IDENTIFIED BY '
    heslo_uzivatele';
mysql> GRANT ALL * TO 'boincadm' IDENTIFIED BY 'heslo_uzivatele';
mysql> GRANT ALL * TO 'boincadm'@'localhost' IDENTIFIED BY 'heslo_uzivatele';
mysql> quit
```

5.9 Konfigurační soubor projektu

Projekt je už skoro připraven, k nasazení zbývá upravit konfigurační soubor projektu podle našich představ. Jednotlivá nastavení tohoto souboru jsme probrali dříve v kapitole 4.1.5, ve výpise (9) uvádíme přímo nastavený konfigurační soubor. Soubor obsahuje i vzorovou konfiguraci daemonů pro aplikaci perm. Tuto aplikaci rozebereme v kapitole 6.

```
<?xml version="1.0" ?>
<boinc>
  <config>
    <upload_dir>
      /project/boinc/upload
    </upload_dir>
    <send_result_abort>
      1
    </send_result_abort>
    <long_name>
      VSB@Home
    </long_name>
    <cgi_url>
      http://158.196.141.75/boinc_cgi/
    </cgi_url>
    <sched_debug_level>
      2
    </sched_debug_level>
    <disable_account_creation>
      1
    </disable_account_creation>
    <uldl_dir_fanout>
      1024
    </uldl_dir_fanout>
    <download_url>
      http://158.196.141.75/boinc/download
    </download_url>
    <log_dir>
      /project/boinc/log_boinc
    </log_dir>
    <app_dir>
      /project/boinc/apps
    </app_dir>
    <download_dir>
      /project/boinc/download
    </download_dir>
    <fuh_debug_level>
      2
    </fuh_debug_level>
    <master_url>
      http://158.196.141.75/boinc/
    </master_url>
    <host>
      boinc
    </host>
    <db_name>
      boinc
    </db_name>
    <shmem_key>
      0x1111d919
    </shmem_key>
    <show_results>
      1
```

```
  </show_results>
  <key_dir>
    /project/boinc/keys/
  </key_dir>
  <upload_url>
    http://158.196.141.75/boinc_cgi/file_upload_handler
  </upload_url>
  <db_host>
    localhost
  </db_host>
  <db_user>
    basic
  </db_user>
  <db_passwd>
    nejake_heslo
  </db_passwd>
  <min_sendwork_interval>
    20
  </min_sendwork_interval>
  <daily_result_quota>
    10000
  </daily_result_quota>
  <one_result_per_host_per_wu/>
  <max_wus_to_send>
    8
  </max_wus_to_send>
 <max_wus_in_progress>
   4
 </max_wus_in_progress>
</config>
<tasks>
  <task>
    <cmd>
      db_dump -d 2 -dump_spec ../db_dump_spec.xml
    </cmd>
    <period>
      24 hours
    </period>
    <output>
      db_dump.out
    </output>
  </task>
</tasks>
<daemons>
  <daemon>
    <cmd>
      feeder -d 1
    </cmd>
  </daemon>
  <daemon>
    <cmd>
      transitioner -d 1
    </cmd>
  </daemon>
```

```
    <daemon>
      <cmd>
        sample_trivial_validator  -d 2 -app perm
      </cmd>
    </daemon>
    <daemon>
      <cmd>
        sample_assimilator -d 2 -app perm
      </cmd>
    </daemon>
    <daemon>
      <cmd>
        file_deleter  -d 2 -app perm
      </cmd>
    </daemon>
    <daemon>
      <cmd>
        perm_work_generator -d 2 -app perm
      </cmd>
    </daemon>
  </daemons>
</boinc>
```

Výpis 9: Konfigurační soubor projektu

Pokud prozatím nemáme vyvinutou žádnou aplikaci, postačí nám ponechat konfigurační soubor v základním vygenerovaném nastavení. Potřebujeme pouze aktualizovat informace týkající se připojení k databázi, tedy doplnit našeho nově vygenerovaného uživatele *basic*.

5.10 Noví uživatelé projektu

Nejdříve zkontrolujeme, zda máme v konfiguračním souboru projektu nastaven element <disable_account_creation> na hodnotu 0. Pokud tomu tak není, nastavení upravíme. Nyní můžeme jednoduše pomocí internetového prohlížeče navštívit domovskou stránku projektu na adrese *http://master_url*. Zde zvolíme odkaz „Vytvořit účet“ a vyplníme požadovaná data. Takto můžeme vytvářet neomezené množství účtů.

V našem případě stačí vytvořit pouze jediný účet, který budeme využívat pro výpočty na všech klientských stanicích.

Po vytvoření účtů jejich registraci opět zakážeme nastavením elementu <disable_account_creation> na hodnotu 1.

5.11 Uvedení projektu do provozu

Nyní máme server nakonfigurován a projekt připraven ke spuštění. Spuštění projektu znamená uvedení jednotlivých daemonů do provozu. Spuštění, zastavení a získání statusu běžících úloh provádíme pomocí těchto příkazů (volání z kořenového adresáře projektu):

- bin/start
- bin/stop
- bin/status

Výše uvedenými příkazy obsluhujeme pouze daemony. Náš projekt má ale i periodické úlohy. Aby mohly být správně spuštěny i periodické úlohy, musíme zavést patřičný záznam do cron tabulky. Záznam nalezneme již vygenerovaný v kořenovém adresáři našeho projektu v souboru s názvem *název_projektu.cronjob* (v našem případě *boinc.cronjob*). Přidání záznamu pro náš projekt vypadá následovně:

```
crontab -e
0,5,10,15,20,25,30,35,40,45,50,55 * * * * /project/boinc/bin/start --cron
```

6 Nasazení aplikace

Server máme připraven, zbývá pouze přidat naši výpočetní aplikaci. Pro náš testovací účely jsme zvolili aplikaci, která provádí testovací komprese pomocí různých permutací.

6.1 Popis aplikace

Naše aplikace se skládá z více spustitelných souborů. Rozhodli jsme se výpočty provádět na platformě Microsoft Windows s procesory typu Intel x86. Všechny části aplikace budeme kompilovat právě pro tuto platformu. Nyní krátce popíšeme jednotlivé části aplikace a jejich funkci.

Program *shuffle.exe* načte soubor s příponou *.perm* (soubor s vygenerovanou permutací) a následně pomocí něj zpřehází obsah testovacího textového souboru (v našem případě soubor *bible.txt*). Výstupní soubor, uložený v pomocném adresáři, je vstupem kompresního programu *BZIP2.EXE*. Ten soubor zkomprimuje. Celý proces se opakuje pro určitý počet vstupních permutačních souborů (například 100). Na závěr zapíšeme výpis pomocného adresáře do textového souboru (v našem případě *results.txt*).

Celá operace pro jeden permutační soubor netrvá příliš dlouhou dobu, proto potřebujeme v jednom work unitu obsluhovat těchto souborů větší množství. Protože se provádí celý cyklus postupně po jednom permutačním souboru, vyvstává nám problém, jak celou proceduru ošetřit pro více permutačních souborů. Zvolili jsme spouštění pomocí skriptu ve spustitelném *.bat* souboru (*run_t.bat*).

6.2 Příprava připojení aplikace k projektu

Abychom mohli naši aplikaci úspěšně připojit k projektu, musíme k ní přidat všechny náležitosti potřebné pro BOINC aplikace.

6.2.1 Vstupní a výstupní šablona pro work unit

V kapitole 4.3.1 jsme uvedli, že není vhodné ve vstupní šabloně definovat příliš veliké množství vstupních souborů. Proto jsme se rozhodli v naší aplikaci přenášet většinu souborů pomocí ZIP archívu. Abychom mohli tento archiv snadno rozbalovat, rozšířili jsme soubor *run_t.bat* o volání programu *unzip.exe*, který budeme spolu se vstupem zasílat na klientskou stanici. Výslednou vstupní šablonu vidíme ve výpise (10). Soubor s touto šablonou se jmenuje *perm_wu.xml* a umístíme jej do adresáře *templates* našeho projektu. Snadno můžeme vidět, že šablona se skládá ze dvou vstupních souborů - archívu *data.zip* a programu *unzip.exe*. Oba sobory se budou kopírovat přímo do *slot* adresáře klientské aplikace.

```
< file_info >
    <number>0</number>
</ file_info >
< file_info >
    <number>1</number>
</ file_info >
```

```
<workunit>
   < file_ref >
      <file_number>0</file_number>
      <open_name>data.zip</open_name>
      <copy_file/>
   </ file_ref >
   < file_ref >
      <file_number>1</file_number>
      <open_name>unzip.exe</open_name>
      <copy_file/>
   </ file_ref >
</workunit>
```

Výpis 10: Vstupní šablona pro work unit aplikace perm

Výstupní šablona pracuje pouze s jedním výstupním souborem (viz. výpis 11). Soubor *results.txt* bude lokálně generován v adresáři results. Jakmile výpočet naší aplikace skončí, bude tento soubor odeslán na server, kde bude přejmenován jménem daného work unitu. Soubor s výstupní šablonou (*perm_result.xml*) umístíme také do *templates* adresáře projektu.

```
< file_info >
    <name><OUTFILE_0/></name>
    <generated_locally/>
    <upload_when_present/>
    <url><UPLOAD_URL/></url>
    <max_nbytes>5000000</max_nbytes>
</ file_info >
<result>
    < file_ref >
        <file_name><OUTFILE_0/></file_name>
        <open_name>./result/results.txt</open_name>
        <copy_file/>
    </ file_ref >
</result>
```

Výpis 11: Výstupní šablona pro work unit aplikace perm

6.2.2 Wrapper

Pro naši aplikaci upravíme zdrojový soubor wrapperu tak, aby načítal jako svůj definiční vstup soubor *perm_config.xml*. Obsah souboru nalezneme ve výpise (12). Nejdříve tedy program *unzip.exe* rozbalí soubor *data.zip* a následně se spustí soubor *run_t.bat*.

```
<job_desc>
   <task>
      <application>unzip.exe</application>
      <command_line>data.zip</command_line>
      <weight>1</weight>
    </task>
    <task>
      <application>run_t.bat</application>
```

```
        <fraction_done_filename>fraction</fraction_done_filename>
        <weight>300</weight>
      </task>
</job_desc>
```

Výpis 12: Definiční soubor pro wrapper

Ve výpise (12) jsme si mohli všimnout definice souboru pro fraction state. Naše dílčí programy fraction soubor nepodporují. Z tohoto důvodu jsme naprogramovali jednoduchý program, který se stará o editaci fraction state v daném souboru. Program je ve spustitelném souboru *fraction_state.exe* a jeho vstupním parametrem je počet kroků. Jeho volání jsme přidali do každé iterace celého cyklu prováděného skriptem *run_t.bat*. Zdrojový kód programu můžeme vidět ve výpise (13).

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char**argv)
{
    FILE *out;

    if (argc < 2)
    {
        cout << "Chyba: uvedte pocet operaci jako parametr\n";
        return(1);
    }
    int count = atoi(argv[1]);
    double step = 1/(double)count;

    ifstream fin("fraction");
    if(fin)
    {
        double d;
        fin >> d;
        d += step;
        out = fopen("fraction", "w");
        fprintf (out, "%f", d);
        fclose(out);
    }else{
        fin.close();
        double d = 0;
        out = fopen("fraction", "w");
        fprintf (out, "%f", d);
        fclose(out);
    }
    return 0;
}
```

Výpis 13: Program pro obsluhu fraction state

Nyní zbývá wrapper zkompilovat, a jeho název přizpůsobit názvu použité platformy a verze v rámci aplikace. Výsledný spustitelný soubor se jmenuje *perm_1.0_windows_intelx86.exe*.

6.2.3 Work generátor

V podkapitole 4.1.3.7 jsme popsali, k čemu work generátor slouží. Pro naši aplikaci potřebujeme generovat vstupní permutační soubory s příponou *.perm*. Aby aplikace neběžela zbytečně krátkou dobu a klient tak nezatěžoval scheduler neustálými dotazy na novou práci, chceme v jednom work unitu zpracovat 100 vstupních permutačních souborů. Protože vstupní permutační soubory budeme generovat v závislosti na čase a nemáme stanoven konečný počet průchodů aplikace, hodilo by se nám soubory generovat přímo za běhu serveru. Další nespornou výhodou tohoto postupu je, že nové work unit se budou generovat pouze pokud počet inicializovaných results na straně serveru klesne pod určitou hodnotu (100).

Náš work generátor vychází z příkladu *sample_work_generator.cpp*, který nalezneme v našich dříve zkompilovaných zdrojových kódech BOINC v adresáři *sched*. Celý zdrojový kód našeho generátoru není pro účel tohoto textu důležitý. Pro nás podstatnou část nalezneme ve výpise (14), celý zdrojový soubor *perm_work_generator.cpp* na přiloženém DVD.

```
// soucasti boinc
#include "boinc_db.h"
#include "error_numbers.h"
#include "backend_lib.h"
#include "parse.h"
#include "util.h"
#include "svn_version.h"
#include "sched_config.h"
#include "sched_util.h"
#include "sched_msgs.h"
#include "str_util.h"
#include "boinc_zip.h"

// udrzuje alespon 100 vysledku
#define CUSHION 100
#define REPLICATION_FACTOR 2

// promenne pro generator
DB_APP app;
char* wu_template;

// vytvarime jeden work unit
int make_job() {
    DB_WORKUNIT wu;
    const char* infiles[2];
    const int ALPHABET_SIZE = 256;
    char path[1024];
    ZipFileList zf;
    int count = 100;
```

```
char fileName[100], fileName2[100];
char* path2 = "../temp_app/perm";
time_t rawtime;
time(&rawtime);

/*
 * zde se nachazi sekce pro generovani permutacnich souboru, pro vetsi prehlednost
 * byla vypustena
 */

// upravime jmena zip balicku podle casoveho razitka
sprintf (fileName, "perm_%d.zip", (unsigned long)rawtime);
sprintf (fileName2, "perm_%d", (unsigned long)rawtime);

// zabalime soubory daneho adresare do archivu a ten umistime do adresare download
config.download_path(fileName, path);
string zipfile = path;
if ( boinc_filelist ("../temp_app/perm", "*", &zf) && zf.size() ) {
    boinc_zip(ZIP_IT, zipfile , &zf);
}
// vymazeme z adresare vygenerovane permutacni soubory
for (int i=0; i<count; i++)
{
    char buf[100];
     sprintf (buf, "%s/%d_%.6d.perm", path2, (unsigned long)rawtime, i);
    remove(buf);
}
// pridame nas archiv jako prvni vstupni soubor daneho work unit
infiles [0] = fileName;
// pridame soubor unzip.exe jako druhy vstupni soubor daneho work unit
config.download_path("unzip.exe", path);
std::ifstream ifs2 ("../temp_app/perm/no_zip/unzip.exe", std::ios::binary);
std::ofstream ofs2(path, std::ios::binary);
ofs2 << ifs2.rdbuf();
infiles [1] = "unzip.exe";
// vymaze vsechny pole pro work unit
wu.clear();
// nastavujeme parametry work unit
strcpy(wu.name, fileName2);
wu.appid = app.id;
wu.min_quorum = 2;
wu.target_nresults = 2;
wu.max_error_results = 5;
wu.max_total_results = 5;
wu.max_success_results = 5;
wu.rsc_fpops_est = 1000000000000;
wu.rsc_fpops_bound = 1000000000000;
wu.rsc_memory_bound = 104857600;
wu.rsc_disk_bound = 786432000;
wu.delay_bound = 7*86400;
// vytvorime work unit
return create_work(
        wu,
        wu_template,
```

```
            "templates/perm_result.xml",
            config.project_path("templates/perm_result.xml"),
            infiles,
            2,
            config
        );
}

// funkce zajistujici generovani work units
void main_loop() {
    int retval;
    // smycka
    while (1) {
        check_stop_daemons();
        int n;
        retval = count_unsent_results(n, app.get_id());
        if (n >= CUSHION) {
        // mame dost results, odpocivame
            sleep(60);
         log_messages.printf(MSG_DEBUG,
                "We have enough jobs now: %d\n", n
            );
        } else {
            // nemame dost result, generujeme nove work unity
            int njobs = (CUSHION-n)/REPLICATION_FACTOR;
            log_messages.printf(MSG_DEBUG,
                "Making %d jobs\n", njobs
            );
            for (int i=0; i<njobs; i++) {
            sleep(1);
                retval = make_job();
                if (retval) {
                    log_messages.printf(MSG_CRITICAL,
                        "can't make job: %d\n", retval
                    );
                    exit(retval);
                }
            }
            // dame transitioneru nekolik sekund na zpracovani
            sleep(5);
        }
    }
}

 void usage(char *name) {
       // funkce popisujici pouziti work generatoru, pro prehlednost bylo telo odstraneno
    );
}

int main(int argc, char** argv) {
    int retval;
    for (int i=1; i<argc; i++) {
        if (is_arg(argv[i], "d")) {
            if (!argv[++i]) {
```

```
                log_messages.printf(MSG_CRITICAL, "%s requires an argument\n\n", argv[--i]);
                usage(argv[0]);
                exit(1);
            }
            int dl = atoi(argv[i]);
            log_messages.set_debug_level(dl);
            if (dl == 4) g_print_queries = true;
        } else if (is_arg(argv[i], "h") || is_arg(argv[i], "help")) {
            usage(argv[0]);
            exit(0);
        } else if (is_arg(argv[i], "v") || is_arg(argv[i], "version")) {
            printf("%s\n", SVN_VERSION);
            exit(0);
        } else {
            log_messages.printf(MSG_CRITICAL, "unknown command line argument: %s\n\n",
                argv[i]);
            usage(argv[0]);
            exit(1);
        }
    }
    retval = config.parse_file();
    if (retval) {
        log_messages.printf(MSG_CRITICAL,
            "Can't parse config.xml: %s\n", boincerror(retval)
        );
        exit(1);
    }
    // pripojeni do databaze
    retval = boinc_db.open(
        config.db_name, config.db_host, config.db_user, config.db_passwd
    );
    if (retval) {
        log_messages.printf(MSG_CRITICAL, "can't open db\n");
        exit(1);
    }
    // hleda aplikaci v databazi
    if (app.lookup("where name='perm'")) {
        log_messages.printf(MSG_CRITICAL, "can't find app\n");
        exit(1);
    }
    // nacte vstupni sablonu pro work unit
    if (read_file_malloc(config.project_path("templates/perm_wu.xml"), wu_template)) {
        log_messages.printf(MSG_CRITICAL, "can't read WU template\n");
        exit(1);
    }
    log_messages.printf(MSG_NORMAL, "Starting\n");
    main_loop();
}
```

Výpis 14: Vybraná část implementace work generátoru

Nejdůležitější funkcí našeho generátoru je funkce **int** make_job(). V této funkci se nejprve provádí generování permutačních souborů do pomocného adresáře */PROJECT/temp_app/perm*. Následně využíváme vnitřní funkce BOINC **int** boinc_zip(**int**

bZipType, **const** std::string szFileZip, **const** ZipFileList* pvectszFileIn), která provede kompresi dat v tomto pomocném adresáři a daný archiv umístí do download adresáře projektu. Permutační soubory jsou smazány a my můžeme nastavit patřičné hodnoty pro náš work unit. Následným zavoláním funkce create_work s danými parametry (objekt DB_WORKUNIT, cesta ke vstupní šabloně aplikace, relativní cesta k výstupní šabloně, absolutní cesta k výst. šabloně, pole cest ke vstupním souborů, počet vstupních souborů, objekt SCHED_CONFIG) vytvoříme jeden work unit.

Funkce **void** main_loop() zajišťuje neustálý chod generátoru. Dotazuje se databáze na počet nerozpracovaných výsledků, pokud je jejich hodnota menší než 50, volá funkci **int** make_job() pro vytvoření dalších work unitů.

6.2.3.1 Kompilace

Jednotlivé knihovny BOINC API jsou natolik provázané, že vyčlenit pouze ty potřebné pro kompilaci generátoru je prakticky nemožné. Z toho důvodu doporučujeme kompilaci provádět následujícím způsobem:

1. Soubor se zdrojovými kódy našeho work generátoru si pojmenujeme jako *sample_work_generator.cpp* a nahradíme jím původní soubor v adresáři sched u našich zkompilovaných souborů BOINC (původní soubor můžeme samozřejmě zálohovat).

2. V hlavním adresáři s našimi zkompilovanými soubory BOINC zavoláme program *make*. Provede se opět kompilace zdrojových kódů BOINC, ale už pouze upravené části, tedy našeho generátoru.

3. Nově zkompilovaný spustitelný soubor přejmenujeme ze *sample_work_generator* na *perm_work_generator*.

4. Přejmenovaný soubor přeneseme do *bin* adresáře našeho projektu.

5. Work generátor *perm_work_generator* je připraven k provozu.

6.2.4 Přesun souborů aplikace

Nejdříve si zrekapitulujeme, které soubory se nachází na správných místech. Soubory pro vstupní a výstupní šablonu (*perm_wu.xml* a *perm_result.xml*) se nachází v adresáři templates našeho projektu. Spustitelný soubor *perm_work_generator* se nachází v našem projektu v adresáři *bin*.

Nyní si vytvoříme v kořenovém adresáři našeho projektu pomocný adresář *temp_app* s podaresářem *perm*. Do adresáře *perm* přidáme soubory *BZIP2.EXE*, *shuffle.exe*, *fraction_state.exe* a *bible.txt*. Tyto soubory budou komprimovány do archivu spolu se vstupními permutacemi. V adresáři *perm* vytvoříme podadresář *no_zip* a do něj vložíme soubor *unzip.exe* (ten bude načítán jako druhý vstupní soubor každého work unitu).

Zbývá vytvořit adresář samotné aplikace. V adresáři *apps* našeho projektu vytvoříme podadresář *perm*, jehož podadresářem bude adresář s názvem

perm_1.0_windows_intelx86.exe (tedy stejným názvem jako má náš wrapper). Do tohoto adresáře vložíme soubory *perm_1.0_windows_intelx86.exe* a *perm_config.xml*.

Nyní jsou všechny soubory na svých místech a vše je připraveno k zavedení naší aplikace.

6.2.5 Editace konfiguračního souboru projektu

Do konfiguračního souboru projektu musíme doplnit nastavení daemonů pro naši aplikaci. Nastavení výsledného konfiguračního souboru bylo už ukázáno ve výpise (9).

6.3 Připojení aplikace

Přidání úplně nové aplikace provádíme pomocí programu *xadd*. Abychom mohli aplikaci přidat, musíme nakonfigurovat soubor *project.xml* v kořenovém adresáři projektu. Výpis (15) zobrazuje tento soubor přímo pro naši aplikaci. Do výpisu jsme v hranatých závorkách přidali na ukázku i další nepovinné elementy. Elementy si popíšeme:

- app - blok pro deklaraci jedné aplikace
- name - zkratka názvu aplikace
- user_friendly_name - plný název aplikace
- min_version - minimální přípustná verze aplikace
- homogeneous_redundancy - výpočty aplikace využívají homogenní redundanci
- weight - váha může upřednostnit výpočty aplikace před výpočty ostatních aplikací
- beta - říká, že se jedná o betatestovanou aplikaci
- platform - blok pro definování jedné platformy (můžeme jich definovat libovolně mnoho)
- name (platform) - zkratka platformy
- user_friendly_name (platform) - plný název platformy

```
<boinc>
    <app>
        <name>perm</name>
        <user_friendly_name>Alphabet Permutation</user_friendly_name>
        [ <min_version>N</min_version> ]
        [ <homogeneous_redundancy>0|1</homogeneous_redundancy> ]
        [ <weight>X</weight> ]
        [ <beta>1</beta> ]
    </app>
    <platform>
        <name>windows_intelx86</name>
```

```
        <user_friendly_name>
    Microsoft Windows (98 or later) running on an Intel x86-compatible CPU
        </user_friendly_name>
      </platform>
</boinc>
```

Výpis 15: Konfigurace souboru project.xml

Aplikaci přidáme spuštěním následujících příkazů z kořenového adresáře projektu:

```
bin/xadd
bin/update_versions
```

Aplikace je nyní připojena k projektu. Pokud budeme chtít přidat novou verzi aplikace, vytvoříme další adresář příslušné verze v adresáři */PROJECT/apps/perm* a provedeme znovu příkaz bin/update_versions. Zbývá pouze restartovat daemony projektu pomocí příkazů:

```
bin/stop
bin/ start
```

Aplikace je plně funkční a můžeme na klienstkých stanicích provádět výpočty. Výsledky nalezneme na serveru v adesáři *sample_results*, který se nachází v kořenovém adresáři projektu.

6.4 Paměťová a výkonová náročnost aplikace

Nasazená aplikace provádí cyklus pro jeden vstupní permutační soubor po dobu přibližně 10-ti sekund. Vzhledem k našemu nastavení replikace se zasílá stejná práce dvakrát, tedy potřebujeme 2 klienstké stanice, abychom provedli jeden výpočet. Pokud však nasadíme na výpočty stanic např. 8, stihneme ideálně spočítat 4-krát více vstupů (pokud mají stanice pouze jednojádrové procesory), než kdybychom prováděli výpočet na jednom počítači. Bohužel vzniká v případě BOINC malá časová latence způsobená přenosem vstupních souborů přes síť.

Zátěž z hlediska operační paměti je pro naši aplikaci zanedbatelná. Během jednoho běhu pro 100 permutačních souborů zabere naše aplikace asi 450 megabajtů prostoru na pevném disku. Po výpočtu je tento prostor okamžitě uvolněn.

7 Instalace klienstské aplikace

Klientská aplikace, která zajišťuje pripojení k projektu, se jmenuje BOINC Manager. Aplikaci je možné instalovat na různých platformách. Pomocí jednoduchého a přehledného rozhranní se můžeme připojit i k několika projektům zároveň. Ihned po instalaci nás aplikace vyzve, abychom se připojili k některémeu projektu. Stačí tedy zadat *http://master_url* našeho projektu a přihlásit se pod jedním z účtů u projektu vytvořených.

7.1 Klasická instalace

Pro základní instalaci na operačním systému Windows stačí stáhnout instalační soubor z adresy *http://boinc.berkeley.edu/download.php* a nainstalovat.

Instalace na operačních systémech Linux se může u každé distribuce lišit. Pro distribuce Fedora, Ubuntu, Debian a i některé další můžeme provést instalaci za pomoci Package Manageru.

7.2 Rozšířená instalace

Můžeme automaticky napojit klienta na projekt pomocí souboru *account_PROJECT.xml*, stačí nainstalovat BOINC Manager na jednom počítači a k našemu projektu se připojit ručně. Tímto se daný soubor pro tohoto uživatele vytvoří. Soubor nalezneme v případě operačního systému Windows v adresáři *C:/Documents and Settings/All Users/Application Data/BOINC*. Soubor stačí zkopírovat do stejného adresáře na jiné klienstské stanici a při příštím spuštění BOINC Manageru dojde k automatickému namapování projektu.

Pokud jsme BOINC Manager instalovali pomocí Package manageru na Linuxovou distribuci Debian nebo Ubuntu, nalezneme soubor v adresáři */var/lib/boinc-client*.

7.3 Vzdálená správa

Připojení k jinému BOINC Manageru provedeme z našeho BOINC Manageru následujícím způsobem:

1. Otevřeme okno našeho BOINC Manageru.
2. Klikneme na tlačítko „*Advanced View*“, které nás přepne do rozšířeného z obrazení.
3. V menu na horní liště zvlolíme *Poktočilé* → *Volba počítače. . .*
4. Vyplníme IP adresu vzdáleného počítače a heslo pro připojení.
5. Nyní nastavujeme vzdálený BOINC Manager stejným způsobem jako náš.
6. Pokud se chceme odpojit, zvolíme *Pokročilé* → *Vypnutí připojeného klienta. . .*
7. Otevře se nám znova okno pro vyplnění IP adresy a hesla. Pokud obě políčka ponecháme prázdná a potvrdíme, BOINC Manager se přepne na náš lokální počítač.

Abychom se mohli ke vzdálené klientské stanici připojit, musíme na ní nejdříve připojení povolit. Toho docílíme editací souborů *gui_rpc_auth.cfg* a *remote_hosts.cfg*. Oba se nachází v adresáři *C:/Documents and Settings/All Users/Application Data/BOINC* (případně */var/lib/boinc-client*).

Soubor *gui_rpc_auth.cfg* obsahuje automaticky vygenerované heslo. Editujeme ho na naše vlastní heslo a soubor uložíme. Toto heslo se bude zadávat při připojování k BOINC Manageru této stanice.

Druhý ze souborů se většinou musí teprve vytvořit a definujeme v něm IP adresy stanic, které se mohou k této stanici připojovat. Na každém řádku souboru je IP adresa jedné stanice.

8 Závěr

Práce se snaží čtenáři snadnou a srozumitelnou formou přiblížit možnosti distribuovaného počítání se softwarovou platformou BOINC. Probírá jednotlivé části platformy a jejich nastavení. Může sloužit jako základní příručka pro vybudování serveru a uvést do problematiky implementace vlastních výpočetních aplikací na základě naší zkušební aplikace. Zvláště užitečný je popis implementace work generátoru, která nemusí být triviální záležitostí. Největším přínosem práce je výsledná VMware image celého serveru, připravená pro okamžité nasazení.

Z celkového pohledu byly probrány především základy nutné pro zprovoznění serveru a umožnění spouštění základních aplikací. Ty by bylo možné v budoucnu rozšířit o problematiku vývoje složitějších aplikací, jako jsou aplikace s grafickým výstupem nebo aplikace podporující technologii CUDA (v současné době je dokumentace k této problematice nedostačující). Dalším možným rozšířením by bylo podrobně popsat editaci a nastavení podpůrné webové prezentace projektu, kterou při současném stavu nepotřebujeme používat. Zajímavým tématem by také mohla být otázka bezpečnosti BOINC serveru.

Kterou problematikou se můžeme v budoucnu zabývat ukáže až nasazení většího množství různorodých projektů na připravený server.

Petr Hanták

9 Literatura

[1] FOSTER, Ian; KESSELMAN, Carl. *The Grid: blueprint for a new computing infrastructure*. San Francisco (California): Morgan Kaufmann, 1999. xxiv, 677 s.

[2] KOSEK, Jiří. *XML pro každého: podrobný průvodce*. Praha: Grada Publishing, 2000. 164 s. ISBN 80-7169-860-1.

[3] *BOINC Trac wiki* [online]. 2007 [cit. 2011-04-15]. Dostupné z WWW: <http://boinc.berkeley.edu/trac/wiki/ProjectMain>.

[4] MYERS, Eric. *Spy Hill .net* [online]. 2010-04-05 [cit. 2011-04-15]. Creating and Configuring a BOINC Project. Dostupné z WWW: <http://www.spy-hill.net/~myers/help/boinc/Create_Project.html>.

[5] BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C. M. *Xml.com* [online]. 1998-02-10 [cit. 2011-04-15]. Annotated XML Specification. Dostupné z WWW: <http://www.xml.com/axml/testaxml.htm>.

A DVD

Přiložené DVD obsahuje tyto základní adresáře:

- ./image
- ./server_stable
- ./server_stable/examples
- ./server_stable/win_buid
- ./text
- ./zdrojove_kody/perm
- ./zdrojove_kody/project

VMware image i s kontrolním součtem se nachází v adresáři *./image*. Image je zkomprimován v archivu *boinc.cs.tgz*. Přístupová hesla nejsou součástí DVD, ale může Vám je poskytnout pan Ing. Jan Platoš Ph.D v kanceláři A1005.

Zkompilované zdrojové kódy pro revizi BOINC č. 22934 jsou k dispozici v adresáři *./server_stable*. Základní ukázkové aplikace se nachází v adresáři *./server_stable/samples* (jejich solution *boinc_samples.sln* pro Microsoft Visual Studio 2008 nalezneme v adresáři *./server_stable/win_buid*).

V adresáři *./text* se nachází text práce v podobě elektronického textového dokumentu (formát PDF). Také jsou přiloženy veškeré soubory použitých obrázků spolu se zdrojovým souborem textu práce pro LaTeX. Adresář obsahuje i použitou verzi makra Diploma, kterou vytvořil pan doc. Mgr. Jiří Dvorský, Ph.D.

Zdrojové kódy a soubory pro chod aplikace *perm* se nachází v adresáři *./zdrojove_kody/perm*. Konfigurační soubor projektu, soubor *project.xml* a další automaticky generované soubory pro nastavení projektu nalezneme v adresáři *./zdrojove_kody/project*.